VISIT…

THE EXPERT'S VOICE® IN JAVA™ TECHNOLOGY

Practical DWR 2 Projects

Explore the design and construction of six complete, rich Web 2.0 applications utilizing DWR, one of the hottest libraries in the Ajax realm today.

Frank W. Zammetti

Foreword by Joe Walker, Creator of DWR

Apress®

Practical DWR 2 Projects

Frank W. Zammetti

Apress®

Practical DWR 2 Projects

Copyright © 2008 by Frank W. Zammetti

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-941-9

ISBN-10 (pbk): 1-59059-941-1

ISBN-13 (electronic): 978-1-4302-0556-2

ISBN-10 (electronic): 1-4302-0556-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Java™ and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc., in the US and other countries. Apress, Inc., is not affiliated with Sun Microsystems, Inc., and this book was written without endorsement from Sun Microsystems, Inc.

Lead Editor: Steve Anglin
Technical Reviewer: Herman van Rosmalen
Editorial Board: Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Kevin Goff, Jonathan Hassell, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Ami Knox
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Dina Quan
Proofreader: April Eddy
Indexer: Julie Grady
Artists: Anthony Volpe, April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com`.

Let's see . . . this is my third book now . . . the first I dedicated to my wife, kids, mom, dad, and John Sheridan for keeping the Shadows off our backs. The second I dedicated to all the animals I've eaten, a batch of childhood friends who helped shape my early life, Denny Crane, and my wife and kids once more. So, I've covered everyone that counts at least once. So, who to dedicate this one to? Who's worthy of my adulation and respect?

Oh, oh! I know . . . ME! I dedicate this book to ME!

OK, fine, I guess I can't do that.

So, I instead dedicate this book to my wife and kids. AGAIN.

I dedicate this book to my sister because I just realized I ***didn't*** *cover everyone that counts, but now I have, sis!*

I dedicate this book to the folks at Harmonix and Bungie because Guitar Hero, Rock Band, and Halo just flat-out rule. I need some Dream Theater, Queensryche, Shadow Gallery, Fates Warning, and Enchant to make my life complete though, so get on that, OK guys?

And lastly, I dedicate this book to all the alien species we have yet to meet. I just hope those dudes need books on web programming, because my kids eat like you wouldn't believe! I'll even take Quatloos!

Contents at a Glance

Foreword xiii
About the Author xv
About the Technical Reviewer xvii
About the Illustrator xix
Acknowledgments xxi
Introduction xxiii

PART 1 Setting the Table

CHAPTER 1 An Introduction to Ajax, RPC, and Modern RIAs 3
CHAPTER 2 Getting to Know DWR 43
CHAPTER 3 Advanced DWR 95

PART 2 The Projects

CHAPTER 4 InstaMail: An Ajax-Based Webmail Client 129
CHAPTER 5 Share Your Knowledge: DWiki, the DWR-Based Wiki 189
CHAPTER 6 Remotely Managing Your Files: DWR File Manager 259
CHAPTER 7 Enter the Enterprise: A DWR-Based Report Portal 329
CHAPTER 8 DWR for Fun and Profit (a DWR Game!) 419
CHAPTER 9 Timekeeper: DWR Even Makes Project Management Fun! 457

INDEX 523

Contents

Foreword xiii
About the Author xv
About the Technical Reviewer xvii
About the Illustrator xix
Acknowledgments xxi
Introduction xxiii

PART 1 Setting the Table

CHAPTER 1 An Introduction to Ajax, RPC, and Modern RIAs 3

A Brief History of Web Development: The "Classic" Model 3
Dawn of a Whole New World: The PC Era 5
Yet Another Revolution: Enter the Web 7
What's So Wrong with the Classic Web? 11
Enter Ajax 14
Why Is Ajax a Paradigm Shift? On the Road to RIAs 18
The Flip Side of the Coin 23
Let's Get to the Good Stuff: Our First Ajax Code, the Manual Way 25
A Quick Postmortem 29
Hey, I Thought This Was Ajax?!? 30
Cutting IN the Middle Man: Ajax Libraries to Ease Our Pain 33
Alternatives to Ajax 36
Hmm, Are We Forgetting Something? What Could It Be?
Oh Yeah, DWR! 39
Summary 40

CHAPTER 2 Getting to Know DWR 43

First Things First: Why DWR at All? 43
DWR: RPC on Steroids for the Web 45
DWR Architectural Overview 47
Getting Ready for the Fun: Your DWR Development Environment 49

A Simple Webapp to Get Us Started 52
Getting the Lay of the Land: Directory Structure 52
From Code to Executable: Ant Build Script 53
Application Configuration: web.xml 54
The Markup: index.jsp 55
On the Server Side: MathServlet.java 56
The Workhorse: MathDelegate.java 58
It's Alive: Seeing It in Action 60
Adding DWR to the Mix 61
The DWR Test/Debug Page 65
Configuring DWR Part 1: web.xml 67
Configuring DWR Part 2: dwr.xml 70
Built-in Creators and Converters 75
The <init> Section 76
The <allow> Section 76
The <signatures> Section 79
Interacting with DWR on the Client 81
Basic Call Syntax 81
Call Metadata Object Approach 82
A Word on Some Funky Syntax 83
Setting Beans on a Remote Object 83
Extended Data Passing to Callbacks 87
Interacting with DWR on the Server 88
DWR Configuration and Other Concepts: The engine.js File 90
Call Batching 92
A Quick Look at util.js, the DWR Utility Package 92
Summary 94

CHAPTER 3 Advanced DWR 95

Locking the Doors: Security in DWR 95
Deny by Default 96
J2EE Security and DWR 98
When Perfection Is Elusive: Error Handling in DWR Applications 101
Handling Warnings 102
Handling Errors 102
Handling Exceptions 102
Edge Cases: Improper Responses 102
The Mechanics of Handling Exceptional Situations 105
Another Word on Exceptions 106
Help from Elsewhere: Accessing Other URLs 107

Turning the Tables: Reverse Ajax 109
Polling 111
Comet 112
Piggybacking 114
The Code of Reverse Ajax 115
Don't Go It Alone: Integration with Frameworks and Libraries 117
Spring 118
JSF 119
WebWork/Struts 2 119
Struts "Classic" 120
Beehive 121
Hibernate 122
Something Old, Something New: Annotations 122
Summary 125

PART 2 The Projects

CHAPTER 4 InstaMail: An Ajax-Based Webmail Client 129

Application Requirements and Goals 129
Dissecting InstaMail 130
Configuration Files 132
The Client-Side Code 134
The Server-Side Code 163
Suggested Exercises 187
Summary 188

CHAPTER 5 Share Your Knowledge: DWiki, the DWR-Based Wiki 189

Application Requirements and Goals 189
FreeMarker 191
Apache Derby 194
Spring JDBC 195
Dissecting DWiki 197
Configuration Files 199
The Client-Side Code 205
The Server-Side Code 230
Suggested Exercises 257
Summary 258

CHAPTER 6 Remotely Managing Your Files: DWR File Manager 259

Application Requirements and Goals 259
dhtmlx UI Components 261
Jakarta Commons IO 268
Jakarta Commons FileUpload 269
Dissecting Fileman 270
Configuration Files 273
The Client-Side Code 278
The Server-Side Code 314
Suggested Exercises 326
Summary 326

CHAPTER 7 Enter the Enterprise: A DWR-Based Report Portal 329

Application Requirements and Goals 329
Spring Dependency Injection (IoC) 331
DataVision 333
Quartz 336
script.aculo.us 337
A Sample Database to Report Against 340
Dissecting RePortal 341
Configuration Files 346
The RePortal Database 351
The Client-Side Code 352
The Server-Side Code 385
Suggested Exercises 416
Summary 417

CHAPTER 8 DWR for Fun and Profit (a DWR Game!) 419

Application Requirements and Goals 419
DWR Annotations 420
Reverse Ajax in Action 421
Anything Else, or Can We Get Goin' Already?!? 422
Dissecting InMemoria 423
Configuration Files 424
The Client-Side Code 426
The Server-Side Code 441
Suggested Exercises 456
Summary 456

CHAPTER 9 Timekeeper: DWR Even Makes Project Management Fun! 457

Application Requirements and Goals 457
HSQLDB 458
Hibernate 459
Ext JS 461
Dissecting Timekeeper 463
Configuration Files 465
The Client-Side Code 471
The Server-Side Code 507
Suggested Exercises 520
Summary 521

INDEX 523

Foreword

The funny thing about getting heavily involved in an open source project is the roller coaster ride you embark on. There's the buzz from seeing the hits to the web server and reading what people think of your project. There's the gnawing feeling of responsibility when you discover very large web sites using your code, and you're worried about bugs you might have created. There's the total flat feeling when a friend tells you he or she is taking your code out of a project because he or she prefers an alternative; and there's the burnout when you just can't keep up with the volume of work and realize that a huge percentage of what you do is not directly development related.

My experiences with open source have opened a huge number of doors. I've met people whom I wouldn't have met otherwise and had job offers that I wouldn't have dreamed of before. There really is a magic buzz to open source.

Marc Andreeson, one of the minds behind Netscape and Ning, wrote recently about how to hire good developers. To paraphrase Marc: "Hire someone that has worked on open source software" (`http://blog.pmarca.com/2007/06/how_to_hire_the.html`).

Some companies rate candidates using trick questions: they get the developers who are good at typing "interview questions" into Google. Some companies rate candidates using industry certifications (MCSD, SCJD, etc.): they get people that work at rich companies that depend on training, and not talent. Some companies rate candidates using CVs/resumes: they end up hiring "talent embroiderers." Some companies rate candidates using interviews: they get the people who sound good and look good.

Unsurprisingly, these selection techniques don't get you the best candidates. So how do you find the developers who love writing good code, who get a buzz from solving the problem in a neat way, and who do take pride in their work?

The answer according to Marc, and according to my experience, is to hire people who love their work enough to get involved with a project that was optional.

So here's your invitation to get a leg up on getting a job with people who hire great developers: get into open source development. It doesn't have to be DWR, although we'd love to have the extra help. Just pick something that excites you and get involved.

The problem with getting started is a typical crossing-the-chasm problem. The first few minutes are easy. You've used a project, liked it, and maybe joined the mailing list. You might even have found something you would like to work on. When you are involved in a project, you know what you are doing and can contribute. But there is a chasm between these places where you are learning the code, learning how the project does things, learning the process, and so on. While you are crossing the chasm, you are unproductive because you are in unfamiliar territory.

So here are a few hints about how to cross the chasm. First, find somewhere that the chasm isn't too wide—start by fixing something small. The chance of any IT project failing is inversely proportional to the size of the project. Start with a simple feature that makes something better. Almost all IT projects have these in abundance.

Second, don't think that because it's tricky, you must be stupid, or that the project must be misguided. There are always reasons why things are tricky. The answer could be historic: when the code was written, people didn't expect the code to be used in this way. Or maybe there is some refactoring that needs doing that hasn't been completed. DWR's code is fairly good because the code is young and we're fanatical about refactoring, but some projects have more history to them.

The difference between those who can cross the chasm and those who can't is drive. You don't need to be a genius, have a brilliant CV, or look good at an interview. Even the ability to type "interview questions" into Google is optional. The people who can cross the chasm are those with the drive to succeed.

Getting involved can come in many forms, and sometimes it's even sort of tangential to the project itself, such as writing a book about the project. Sometimes the tangential help is some of the most valuable. The things developers leave out are often things they are bad at. For years, I've wanted there to be a DWR book but known I'm the wrong person to write it, so I'm particularly pleased to see Frank step forward to write the first DWR book. Thanks for having the drive to get involved, Frank.

Joe Walker
Creator of DWR

About the Author

FRANK W. ZAMMETTI is a developer/architect/whatever is called for at any particular moment in time for a large mutual fund servicing company in the United States by day, and a multi-project open source contributor by night. And, after three books, he can probably finally say "author by night" legitimately too!

Frank has been involved, in one form or another, with computers for 25+ years, which is about 75 percent of his life thus far (only counting the natural part anyway) and has been programming right from the start. When other kids were out playing tag and riding bikes, Frank was . . . well, he was out there with them because his mom wouldn't let him sit in the house all day. But at night, ah at night, Frank was hacking away at code all through the Hour of the Wolf, when he wasn't blowing up his dad's workbench with some wacky electronics contraption at least. About 15 of those 25 years have been "professional" years, which simply means he was (and still is) being paid to pretend he knows what he's talking about.

Frank has written two other books which have garnered rave reviews. One guy named Bob said of his first book, "Frank's writing is more tolerable than most," and Jimbo McMalmun from Piedmont wrote, "Of all the books on this topic I've read, Frank's was the first." Lastly, *Online Computer Magazine of America* said: "The bright yellow cover made us think of bees, which is the same grade we give Mr. Zammetti's efforts." Seriously folks, you flatter me!

Frank lives in Pennsylvania with his wife of 13 years, his two children Andrew and Ashley, his dog Belle, and two guinea pigs that apparently have not heard the phrase "Never bite the hand that feeds you." When Frank isn't writing or coding, he can most usually be found getting his rear end handed to him by Andrew in Guitar Hero, or trying to get yet another game to work on Ashley's outdated PC (she's only four and yet already knows more about computers than many of the adults Frank has met over the years!).

Frank has a deep set of personal beliefs that guide his life. Among them are the belief that the Daleks are redeemable, the belief that tacos were handed down from God herself, and the belief that *Friends* is singularly the most overrated show of all time.

And in case you haven't guessed by now, Frank truly believes that next to a healthy colon, little in life is more important than laughter, so in all seriousness, he hopes you have a good time reading this book **as well as** learn a thing or two!

About the Technical Reviewer

HERMAN VAN ROSMALEN works as a developer/software architect for De Nederlandsche Bank N.V., the central bank of the Netherlands. He has more than 20 years of experience in developing software applications in a variety of programming languages. Herman has been involved in building mainframe, PC, and client-server applications. Since 2000, however, he has been involved mainly in building J2EE web-based applications. After working with Struts for years (pre-1.0), he got interested in Ajax and joined the Java Web Parts open source project in 2005. Besides this book, Herman has also served as technical editor for the Apress titles *Practical Ajax Projects with Java Technology* and *Practical Javascript, DOM Scripting, and Ajax Projects*. Herman lives in a small town, Pijnacker, in the Netherlands with his wife Liesbeth and their children Barbara, Leonie, and Ramon. You can reach him via e-mail at herros@gmail.com.

About the Illustrator

ANTHONY VOLPE did the illustrations for this book. He has worked on several video games with author Frank Zammetti, including Invasion: Trivia!, IO Lander, K&G Arcade, and Ajax Warrior. Anthony lives in Collegeville, Pennsylvania, and works as a graphic designer and front-end web developer. His hobbies include recording music, writing fiction, making video games, and going to karaoke bars to make a spectacle of himself.

Acknowledgments

There are quite a few people who played a role in making this book happen, so I'll do my best to remember everyone, but I probably won't, so apologies in advance.

First and foremost, I'd like to thank Joe Walker, creator of DWR, for, well, **creating DWR**! I'd also like to thank him for his support in writing this book, writing the foreword, and asking me to copresent a session with him at The Ajax Experience in Boston (next time we can actually rehearse and I can earn my pay!).

I'd like to acknowledge all the folks at Apress who worked on this book and gave me a third authoring experience that was an absolute pleasure. Beth Christmas, Ami Knox, Steve Anglin, Kelly Winquist, April Eddy, and all the folks I don't even know by name, thank you!

Anthony Volpe once again worked his graphic/art magic on the illustrations in this book and even on some diagrams this time around. Thanks, man!

I have to, of course, throw the usual thanks at Herman Van Rosmalen for again taking on technical review duties for me. After doing a technical review on Ian Roughley's book *Practical Apache Struts 2 Web 2.0 Projects* myself, I now understand exactly the effort that work involves, and you've been with me for three books now, and your efforts have never been appreciated more, my friend!

Last but certainly not least, I'd like to thank **YOU** for buying and reading this book! A book without a reader is like a donut without a police officer (I kid, I love the folks in blue!), peanut butter without chocolate, or a politician without a scandal: some things just can't exist without the other!

As I said, I know I'm almost certainly forgetting to acknowledge someone here, so how about I just thank anyone who has ever, is now, or will ever draw breath and be done with it? If I had the technology, I could be like Wowbagger the Infinitely Prolonged, but handing out "Thanks!" instead of insults. Alas, I've been busy writing this book so have not yet completed my Grand Unified Theory that would allow for that, so this'll have to do!

Introduction

DWR. Three little letters that possess more power than General Zod under a yellow sun. Three little letters that amount to a whole book's worth of text.

You know, this is my third authoring gig, and the first two were kind of similar in that they covered a variety of libraries and toolkits, and all sorts of different approaches to Ajax, RIA, and Web 2.0 development. This time around though, I'm writing about a single topic, and when I was first asked to do it, I was a little worried frankly . . . Was there enough to write about? Would I get bored? How much different would it be to stay, essentially, on one topic for 500+ pages?

If it had been any other library, I'm not sure what the answer would be, but with DWR, as it turns out, it was both easy and enjoyable the whole way through!

You see, DWR makes developing advanced webapps almost **too** easy sometimes! It makes it effortless to do some things that can at times be difficult to do, and of course do well. It puts so much power in the palm of your hand without it being a heavy lift that you sometimes have to stop and realize just what kind of magic it must be doing under the covers to make it look so easy at the level you interact with it on.

Now, one way in which this book is just like my previous two is that it takes a very pragmatic approach to learning. I know personally, I need to see things in action, and more than that, I need it to be explained to me. Don't just throw code in front of me; that's often more trouble than it's worth. Instead, put code in front of me that's commented well, that's constructed in a consistent manner, and that has some explanation to go along with it. For me, that's how I learn best, and I know I'm not alone. Also, don't waste my time with contrived, overly simplistic examples that don't go deep enough to be of any real use. No, show me a real, working, practical application and tear it apart, that's what I like. And the fact that someone has been willing to publish three such books from me proves there must be like-minded people out there!

Another way this book is like my previous efforts is that I have a guiding philosophy that says that life is pretty darned tough under the best of circumstances, so why not lighten the load a little and bring laughter to things any chance we get? That comes through in my writing. My friends and coworkers would tell you that I'll make a joke about just about anything at just about any time. It's my way, for better or worse (and I'd be lying if I didn't say it was for worse sometimes!). That's also the way I write. The greatest compliment I've been paid, a couple of times by various folks, is that I write like I'm sitting there speaking to you. It's not on purpose; it's just the way I naturally write. Enough people seem to like it and think it works to allow me to keep getting books published, so I surmise it's not a bad thing.

That's what this book is: a series of practical applications, torn apart and described, in a (hopefully!) entertaining style that will, with a little luck, keep you smiling as you learn about a truly great library in DWR.

2ca983ba3745582e6151dc1b079b2db0

An Overview of This Book

This book is broken down into two sections. The first section, which consists of the first three chapters, is the more "academic" section. Here's a rundown of the chapters:

- Chapter 1 is an introduction to Ajax. It covers the history of application development, how it's evolved onto the Web, and how Ajax is used in modern webapp development. It even touches on some basic Ajax code.
- Chapter 2 formally introduces DWR, describing what it is, how it works (at a high level), and gets into some basic usages of it.
- Chapter 3 goes into more depth on DWR, including more advanced features like reverse Ajax.

After this section comes the second section, which is the projects. Here we have a series of six chapters presenting one application in each:

- Chapter 4 is where we develop InstaMail, an Ajax-based webmail client.
- Chapter 5 shows how to develop a wiki application using DWR. This chapter includes usage of the Freemarker templating library and the Derby database, a pure Java embeddable database.
- Chapter 6 shows how to develop a decent file manager application, à la Windows Explorer. It uses the dhtmlx GUI widget components to make a good-looking and highly functional user interface, as well as various Jakarta Commons libraries.
- Chapter 7 is the chapter in which we develop an enterprise reporting portal. This application utilizes the DataVision open source reporting tool, as well as script.aculo.us for UI effects, Spring for database access, and Derby once again for a data store.
- Chapter 8 shows how to build a game using DWR. I admit it's not going to rival World of Warcraft or anything, but it *does* show how reverse Ajax, sometimes called Comet, can be used with DWR with little effort.
- Chapter 9, the final chapter, is where we construct a time-tracking system for managing projects and booking time. This project uses the fantastic Ext JS library to create a UI that has a very native feel to it. This project also uses Hibernate for data access and HSQLDB for its data store.

As you can see, the projects cover a lot of territory and mix in a number of supporting libraries that, when combined with the power of DWR, make for some pretty decent applications without having to kill ourselves writing every last bit of code.

Obtaining This Book's Source Code

One thing that's important to me, and this may sound a little gross at first, is that you should be able to, by and large, read this book in the bathroom. What I mean is, you generally shouldn't **have** to be sitting in front of a computer to get the full benefit from this book. To that end, I try to show as much of the code as possible, as well as screenshots of each application, enough so that if you never play with the real thing, it won't kill the experience entirely.

That being said, when you write a book, you sometimes in fact do have to make sacrifices and not show everything. So, even though I strive to make this book "bathroom-ready™," it will be assumed at various times that you have in fact downloaded all the source code and are running the application or can refer to the full source that won't be on the printed page. To that end, the source code for all the projects in this book, as well as WAR files you can drop in your servlet container of choice immediately to run, can be downloaded from the Apress web site at `www.apress.com`. Click the Source Code/Download link and then find *Practical DWR 2 Projects* in the list. From the book's home page, you can download a single ZIP file that contains everything, all code, WAR files, and anything else that may be required. Each chapter has its own directory, so it's easy to find what you need at any point.

Obtaining Updates for This Book

Look, I'm purfect, and hence you willn't find not a sinngle error anywhare in this bok.

AHEM.

Seriously though, writing a book is a lot of hard work and late, tired nights when you have a full-time job and two kids after that. Even with a diligent author, a competent technical reviewer, an editor, copy editor, layout editor, project manager, and maybe others, mistakes are bound to creep in. I will therefore apologize in advance and beg your forgiveness for anything you may find!

That being said, a current errata list is available on this book's home page on the Apress web site, along with information on how to submit reports of anything you find. I very much appreciate any such feedback you would like to offer and want to assure you that as thanks, I will put in a good word for you in the afterlife (of course, you'd better hope I go to the right place!).

Contacting the Author

If for some bizarre reason you would like to contact me, feel free to e-mail me at `fzammetti@omnytex.com`. I loathe spam though, so if you're trying to sell me any sort of "medication," want to refinance my mortgage, have some pictures I **have** to see, or otherwise just want to annoy me, please note that I retain the right to launch a small nuclear volley in your general direction (and as powerful as DWR is, I wouldn't be surprised if control of a nuclear arsenal is a feature I just didn't notice!).

PART 1

Setting the Table

First learn computer science and all the theory. Next develop a programming style. Then forget all that and just hack.

—George Carrette

Writing code has a place in the human hierarchy worth somewhere above grave robbing and beneath managing.

—Gerald Weinberg

Real Programmers always confuse Christmas and Halloween because Oct31 == Dec25.

—Andrew Rutherford

A computer is like an Old Testament god, with a lot of rules and no mercy.

—Joseph Campbell

In theory, there is no difference between theory and practice. But in practice, there is.

—Yogi Berra

Mister, are you a computer biologist?

—An unnamed preschool student, said to me as I worked on the classroom PC

The Internet is a telephone system that's gotten uppity.

—Clifford Stoll

I got an idea, an idea so smart my head would explode if I even began to know what I was talking about.

—Peter Griffin, from the *Family Guy* episode "Lethal Weapons"

CHAPTER 1

An Introduction to Ajax, RPC, and Modern RIAs

If this is your first experience with Ajax, and even web development in general, this chapter will serve as a good introduction to get you up to speed for what is to come. If, however, you are a relatively experienced developer, and especially if Ajax is not new to you, feel free to skip most of this chapter, as it will likely be just a review for you (although I do suggest reading the section introducing DWR near the end, since that's presumably why you're here!). This chapter explores the wonderful world of Ajax by examining how applications in general, web applications in particular, have been developed over the past decade and a half or so. You'll discover an interesting cycle in terms of the basic structure of applications. We'll look at our first example of Ajax in action and talk about why Ajax is important and how it can fundamentally alter how you develop applications. We'll also briefly touch on some of the alternatives to Ajax, and some of the existing libraries and toolkits that make Ajax easier. Last but not least, I'll give you a first quick look at DWR, just to whet your appetite for what is to come later.

A Brief History of Web Development: The "Classic" Model

In the beginning, there was the Web. And it was good. All manner of catchy new words, phrases, and terms entered the lexicon, and we felt all the more cooler saying them. (Come on, admit it, you felt like Spock the first couple of times you used the word "hypertext" in conversation, didn't you?) *Webapps*, as our work came to be known, were born. These apps were in a sense a throwback to years gone by when applications were hosted on "big iron" and were accessed in a timeshare fashion. They were in no way, shape, or form as "flashy" as the Visual Basic, PowerBuilder, and C++ *fat clients* that followed them (which are still used today, of course, although less and less so with the advent of webapps).

In fact, if you really think about it, application development has followed a very up-and-down pattern, and if you walk down the timeline carefully and examine it, this pattern begins to emerge. Starting with what I term the "modern" era, that is, the era in which applications took a form that most of us would basically recognize, we first encounter simple terminal emulation applications (for the sake of this argument, we'll skip the actual terminal period!) used to access remotely running processes. Screens like the one shown in Figure 1-1 were typical of those types of applications.

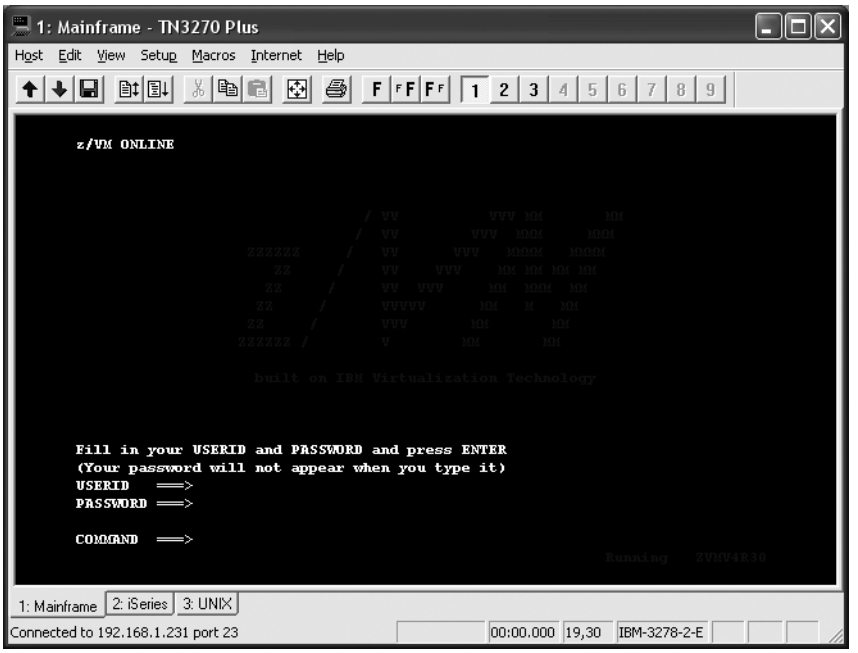


Figure 1-1. *3270 mainframe "green screen" terminal display*

3270 screens are, of course, completely relevant in the sense that they are still used quite a bit, especially in the business world, as anyone who has done any sort of mainframe work can attest to. There are two interesting things to note, for the sake of this discussion. First, notice the simple nature of the user interfaces (UIs) back then. They were text-only, usually limited to 80 columns by 25 lines of text, with extremely limited data entry capabilities, essentially just editable mapped regions. Things like drop-down menus, check boxes, and grids were completely unknown in this domain. If it was a well-written application, you would be fortunate and have a real menu like so:

C. Create record

D. Delete record

E. Edit record

If you were unlucky, you would just have something like this:

..... 01A7C0D9ABABAC00

..... 89A6B3E34D79E998

If you have never worked on a mainframe, let me briefly explain what that is. For editing files (called *data sets*) on a mainframe, you usually use a tool called *TSO/ISPF*. This is just a form of text editor. This editor can be flipped between textual display and hex display, and the preceding is the hex display version. The dots that start each line make up the command area.

For instance, to insert a line above the line that begins with 89, you would go to the first dot in that line and replace it with i, then press the Enter key. If you wanted to delete that line, plus the line that starts with 01, you would go to the line that starts with 01, type dd over the first two dots, then go to the line you just inserted and put dd there as well, then press Enter (dd is for deleting a block of lines; you can use a single d to delete a single line).

MORE ON 3270, FOR YOU HISTORY BUFFS OUT THERE

TN3270, or just plain 3270 as it is commonly called, was created by IBM in 1972. It is a form of *dumb terminal*, or *display device*, usually used to communicate with IBM mainframe systems. Because these displays typically only showed text in a single color, namely green, they have come to be informally referred to as *green screens*. 3270 was somewhat innovative in that it accepted large blocks of data known as *data-streams*, which served to minimized the number of I/O interrupts required, making it (at the time) a high-speed proprietary communication interface.

3270 terminals, the physical entities that is, have not been manufactured for a number of years; however, the 3270 protocol lives on to this day via emulation packages (Attachmate being a big one in the business world). Green screens can still be seen around the world though, most notably in call centers of many large businesses, where they are still the tool of choice . . . some would say tool of inertia. In any case, they are still used in many places because, ironically, these interfaces tend to be more productive and efficient than competing (read: web-based) interfaces. Sometimes, simplicity really is king!

Second, and more important here, is the question of what happens when the user performs an action that requires the application to do something. In many cases, what would happen is that the mainframe would redraw the entire screen, even the parts that would not change as a result of the operation. Every single operation occurred on the mainframe, and there was no local processing to speak of. Not even simple input validation was performed on the client; it was simply a view of a remote application's state, nothing more.

Dawn of a Whole New World: The PC Era

With the advent of the PC, when the amount of local processing power advanced orders of magnitude, a new trend emerged. At this point we began to see applications hosted locally instead of on central mainframes where at least some portion of the application actually executed locally. Many times, the entire application itself was running on the machine that the user was using. With the growth in popularity of Microsoft Windows more than anything else (I know, some will cringe at the thought of crediting Microsoft with **anything**, but there's no sense denying the reality of things), fat clients, as they came to be known, were suddenly the de facto standard in application development. The UI available in this paradigm was immensely more powerful and user-friendly, but the central hardware faded in importance for the most part (things like database servers notwithstanding). Screens like the one in Figure 1-2 became the norm.

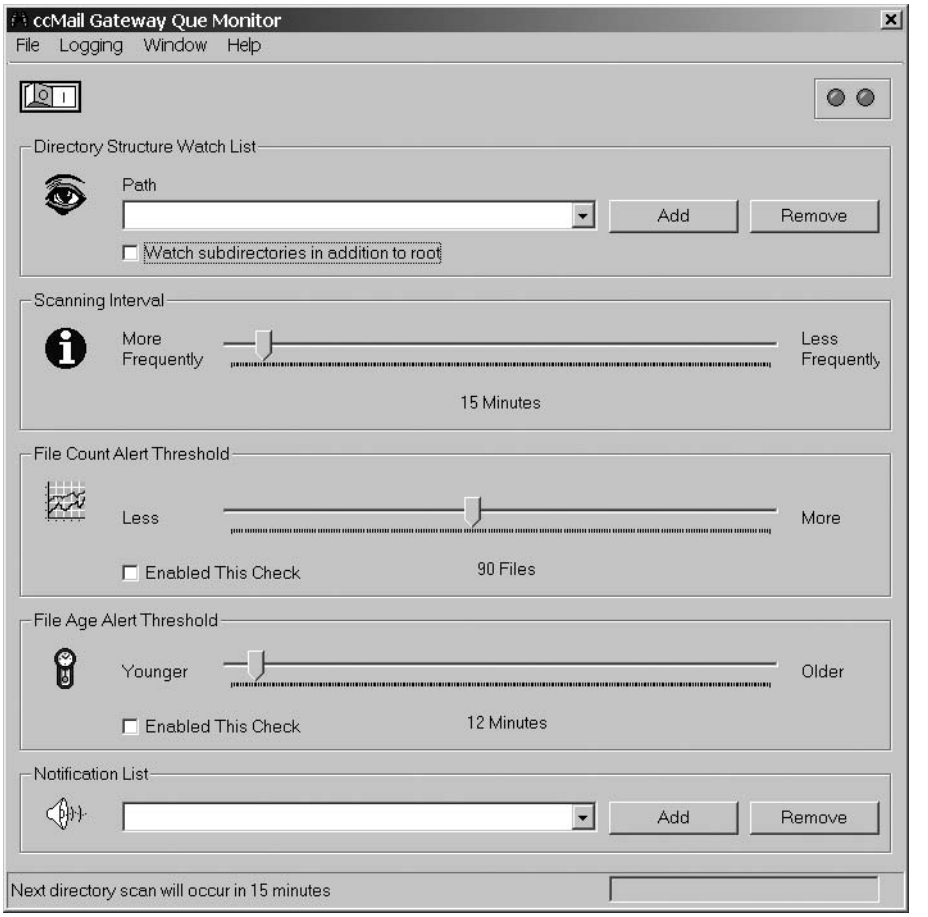


Figure 1-2. *A typical fat-client application*

More for my own ego than anything else, I'll tell you that this is a Visual Basic application I wrote pretty close to ten years ago. ccMail, which is a file-based e-mail system seldom used today but quite popular years ago, works by storing messages on a file system in queues. For a while, the IT guys at my company had an issue with messages getting "stuck" in queues. As you can imagine, that's not supposed to happen; they are supposed to hit a queue, and after some relatively brief period of time (a minute or two at most) get forwarded to their destination. Instead of finding the root cause of the problem, we got creative and wrote this application to check the queues for messages that were too old, or for an overabundance of messages in a queue, and send e-mail notifications alerting someone to the situation. Fortunately, we stopped using ccMail a short time after that, so it didn't matter that we never found the real problem.

Note how much richer the available UI metaphors are. It should come as no surprise to you and the rest of the user community out there that this is universally seen as "better" (you **do** see this as better, don't you?). Better is, of course, a relative term, and in some cases it is not better. You might think that people doing heads-down data entry all day might actually prefer those old green screens more because they lend themselves to more efficient keyboard-based data entry, and you would be right in many cases. No fussing with a mouse. No pointing and clicking. No need to take their eyes off the document they are keying off of to save a record. While all of that is true, it cannot be denied that, by and large, people will choose a fat-client version of a given application over a text-only version of it any day of the week. Go ahead, try it, I dare you! Take an existing mainframe application and put it side by side with a fat-client version of the same application and see how many users actually want the mainframe version. I'll give you a hint: it will be less than one, but not negative (although you may have to make the fat-client version "keying friendly" to make them happy, but that's another point).

Thinking about how the application actually functions, though, what happens here when the user clicks a button, for example, or slides a slider, or clicks a menu item? In most cases, only some region of the screen will be updated, and no external system is interacted with (usually). This is obviously more efficient and in all likelihood more user-friendly as well.

Yet Another Revolution: Enter the Web

But what happened next in our timeline? A bit of a monkey wrench got thrown in the works with the rise of the Internet, and more specifically, the World Wide Web component, or just the Web for short (remember, the Web is not, in and of itself, the Internet!). With the emergence of the Web, screens like the one in Figure 1-3 became commonplace.

Wait a second, what happened? Where did all our fancy radio buttons, 3D buttons, list boxes, and all that go? The first iteration of the Web looked a heck of a lot visually like the old mainframe world. More important, though, is what was happening under the hood: we went back to the old way of doing things in terms of centralized machines actually running the applications, and entire screens at a time being redrawn for virtually every user interaction.

In a very real sense, we took a big step backward. The screen was redrawn by the server and returned to the user with each operation (amazingly, a paradigm that is still very much with us today, although if you're reading this book, you likely see the logic in changing that, and we'll do our part to make that happen!). Each and every user interaction (ignoring client-side scripting for the moment because it was not immediately available to the first web developers) required a call to a server to do the heavy lifting. See, we went back to the mainframe way of doing things, more or less! We didn't all just lose our minds overnight, of course; there were some good reasons for doing this. Avoidance of *DLL Hell*, the phenomenon in the Windows world where library versions conflict and cause all sorts of headaches, was certainly one of them, because that is even today a common complaint about fat clients on the Windows platform (newer versions of Windows alleviate it somewhat, but not entirely).

Another reason was the need to distribute applications. When an application runs on a centralized server, it can be accessed from any PC with a web browser without having to first install it. Another good reason was the relative ease of application development. At least in the beginning when webapps were fairly simple things done with little more than HTML and simple back-end CGI programs, almost anyone could quickly and easily pick it up. The learning curve was not all that high, even for those who had not done much application development before.

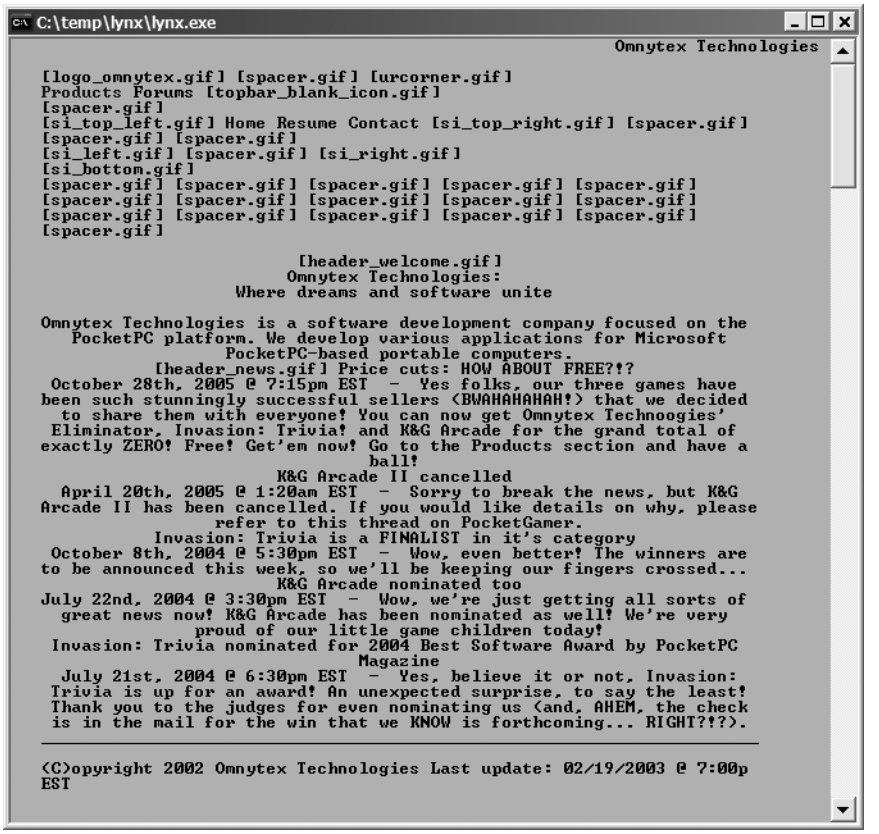

C:\temp\lynx\lynx.exe
Omnytex Technologies
[logo_omnytex.gif] [spacer.gif] [urcorner.gif]
Products Forums [topbar_blank_icon.gif]
[spacer.gif]
[si_top_left.gif] Home Resume Contact [si_top_right.gif] [spacer.gif]
[spacer.gif] [spacer.gif]
[si_left.gif] [spacer.gif] [si_right.gif]
[si_bottom.gif]
[spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif]
[spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif]
[spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif] [spacer.gif]
[spacer.gif]
[header_welcome.gif]
Omnytex Technologies:
Where dreams and software unite
Omnytex Technologies is a software development company focused on the PocketPC platform. We develop various applications for Microsoft PocketPC-based portable computers.
[header_news.gif] Price cuts: HOW ABOUT FREE?!?
October 28th, 2005 @ 7:15pm EST - Yes folks, our three games have been such stunningly successful sellers (BWAHAHAHAH!) that we decided to share them with everyone! You can now get Omnytex Technoogies' Eliminator, Invasion: Trivia! and K&G Arcade for the grand total of exactly ZERO! Free! Get'em now! Go to the Products section and have a ball!
K&G Arcade II cancelled
April 20th, 2005 @ 1:20am EST - Sorry to break the news, but K&G Arcade II has been cancelled. If you would like details on why, please refer to this thread on PocketGamer.
Invasion: Trivia is a FINALIST in it's category
October 8th, 2004 @ 5:30pm EST - Wow, even better! The winners are to be announced this week, so we'll be keeping our fingers crossed...
K&G Arcade nominated too
July 22nd, 2004 @ 3:30pm EST - Wow, we're just getting all sorts of great news now! K&G Arcade has been nominated as well! We're very proud of our little game children today!
Invasion: Trivia nominated for 2004 Best Software Award by PocketPC Magazine
July 21st, 2004 @ 6:30pm EST - Yes, believe it or not, Invasion: Trivia is up for an award! An unexpected surprise, to say the least! Thank you to the judges for even nominating us (and, AHEM, the check is in the mail for the win that we KNOW is forthcoming... RIGHT?!?).
(C)opyright 2002 Omnytex Technologies Last update: 02/19/2003 @ 7:00p EST

Figure 1-3. *The Omnytex Technologies site (*`www.omnytex.com`*) as seen in Lynx, a text-based browser*

MORE ON CGI

CGI, which stands for Common Gateway Interface, is a protocol used to interface an external software application with some information server, typically a web server. This mechanism allows for passing of requests from the client, a web browser in the case of a web server, to the server. The server then returns the output of the CGI application execution to the client.

CGI began its life based on discussions on the mailing list www-talk involving many individuals including Rob McCool. Rob was working at NCSA (National Center for Supercomputing Applications), and he wrote the initial CGI specification based on those discussions and also provided the first "reference" implementation for the NCSA HTTPd web server (which, after morphing a lot and being renamed, became the ubiquitous Apache web server we all know and love today). This implementation, and in fact most implementations ever written (although not all), used system environment variables to store the parameters passed in from the client and spawned a new thread of execution for executing the CGI application in, which as you can guess is highly inefficient. Later incarnations got around this problem in various creative ways.

CGI is still seen today, although far less often than before.

Of course, the Web grew up mighty quick! In what seemed like the blink of an eye, we moved from Figure 1-3 to Figure 1-4.

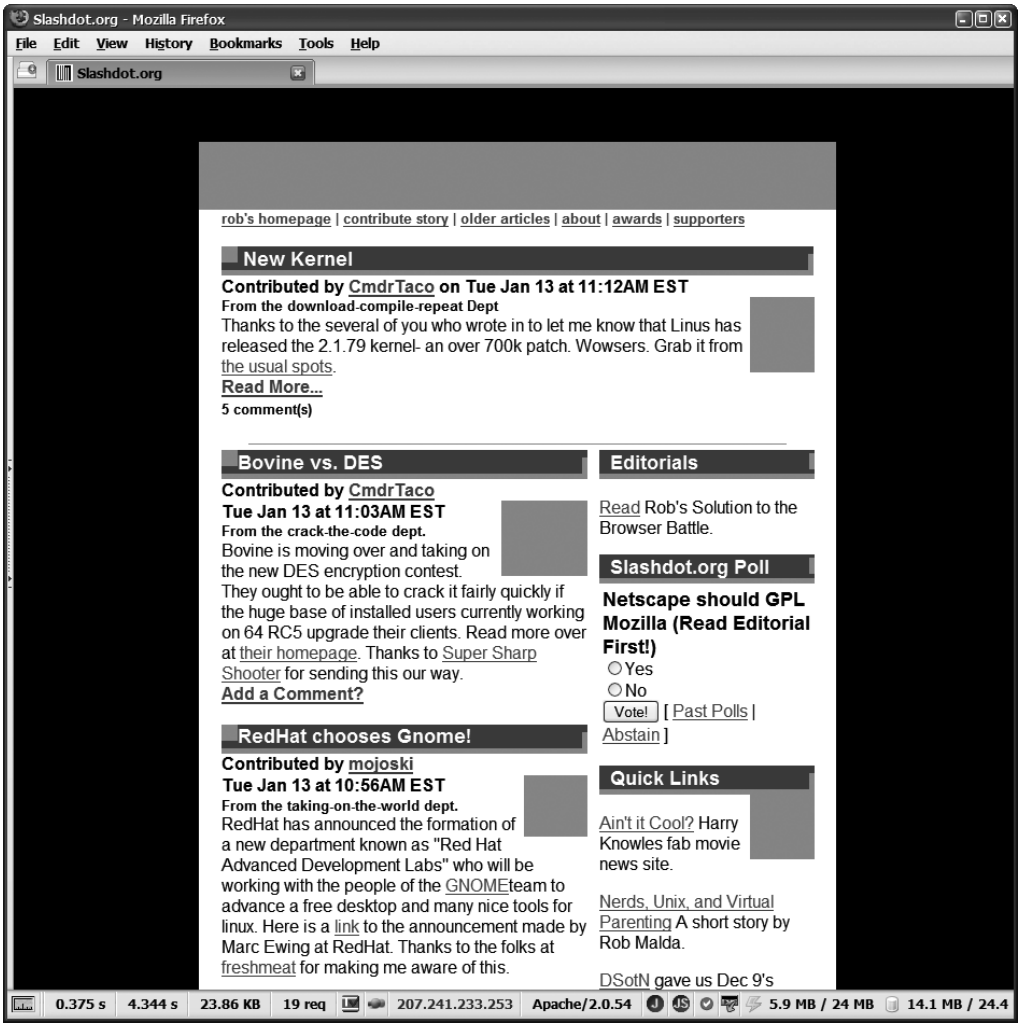

Slashdot.org - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Slashdot.org
rob's homepage | contribute story | older articles | about | awards | supporters
New Kernel
Contributed by CmdrTaco on Tue Jan 13 at 11:12AM EST
From the download-compile-repeat Dept
Thanks to the several of you who wrote in to let me know that Linus has released the 2.1.79 kernel- an over 700k patch. Wowsers. Grab it from the usual spots.
Read More...
5 comment(s)
Bovine vs. DES
Contributed by CmdrTaco
Tue Jan 13 at 11:03AM EST
From the crack-the-code dept.
Bovine is moving over and taking on the new DES encryption contest. They ought to be able to crack it fairly quickly if the huge base of installed users currently working on 64 RC5 upgrade their clients. Read more over at their homepage. Thanks to Super Sharp Shooter for sending this our way.
Add a Comment?
RedHat chooses Gnome!
Contributed by mojoski
Tue Jan 13 at 10:56AM EST
From the taking-on-the-world dept.
RedHat has announced the formation of a new department known as "Red Hat Advanced Development Labs" who will be working with the people of the GNOMEteam to advance a free desktop and many nice tools for linux. Here is a link to the announcement made by Marc Ewing at RedHat. Thanks to the folks at freshmeat for making me aware of this.
Editorials
Read Rob's Solution to the Browser Battle.
Slashdot.org Poll
Netscape should GPL Mozilla (Read Editorial First!)
Yes
No
Vote! [Past Polls | Abstain]
Quick Links
Ain't it Cool? Harry Knowles fab movie news site.
Nerds, Unix, and Virtual Parenting A short story by Rob Malda.
DSotN gave us Dec 9's
0.375 s 4.344 s 23.86 KB 19 req 207.241.233.253 Apache/2.0.54 5.9 MB / 24 MB 14.1 MB / 24.4

Figure 1-4. *Slashdot, circa 1998*

Now, that certainly looks a bit better, from a visual standpoint for sure. In addition to just the visual quality, we had a more robust palette of UI widgets available like drop-down menus, radio buttons, check boxes, and so forth. In many ways it was even better than the fat clients that preceded the rise of the Web because *multimedia* presentation was now becoming the norm. Graphics started to become a big part of what we were doing, as did even sound and video eventually, so visually things were looking a lot better.

What about those pesky user interactions, though? Yes, you guessed it: we were still redrawing the entire screen each time at this point. The beginnings of client-side scripting emerged though, and this allowed at least some functionality to occur without the server, but

by and large it was still two-tier architecture: a view tier and, well, the rest! For a while we had frames too, which alleviated that problem to some degree, but that was really more a minor divergence on the path than a contributor to the underlying pattern.

Before you could say "Internet time," though, we found ourselves in the "modern" era of the Web, or what we affectionately call "today" (see Figure 1-5).

Omnytex Technologies - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Omnytex Technologies
Omnytex Technologies
Products
Forums
Products
Electro Version 1.2
Product Info Reviews Support Updates Trial
$25.00 Purchase
Electro is the premiere electronics reference and tool collection for the PocketPC platform. With numerous calculators, reference tables and other tools for the electronics technician as well as the electrical engineer, Electro is one PocketPC program you cannot do without!
K&G Arcade Version 1.6
Product Info Reviews Support Updates
FREE! Download
K&G Arcade is unlike any other PocketPC game out there... Imagine being beamed aboard an alien spacecraft by two wise-cracking aliens from the future. Further imagine being forced to try and escape their ship by exploring five maze-like levels and playing 25 mini-games throughout, all the while avoiding deadly robots and conversing with strange fellow captives. Sound like a good time? You bet it is! That's K&G Arcade in a nutshell! And best of all, this game is totally, completely, 100% FREE!!
Invasion: Trivia Version 1.6
Product Info Reviews Support Updates
FREE! Download
Trivia, with a very sick twist! Earth is threatened by two wise-cracking aliens and you are choosen to defend your world in a contest of knoweldge! Krelmac and Gentoo invite you to fight for the survival of your world in a game of knowledge and wits. DIMWITS that is! And best of all, this game is totally, completely, 100% FREE!!
0.250 s 1.344 s 26.72 KB 11 req 216.227.215.170 Apache 1.2 MB / 24 MB 11.9 MB / 24.4 MB

Figure 1-5. *The "modern" Web, the Omnytex Technologies site as the example*

So, at least visually, as compared to the fat clients we were largely using throughout the late '80s and early '90s, we are now at about the same point, and maybe even a bit beyond that arguably. What about the UI elements available to us? Well, they are somewhere in the middle.

We have radio buttons and check boxes and drop-down menus and all that, but they are not quite as powerful as their fat-client counterparts. Still, it is clearly better than the text regions we had before the Web.

The term *Internet time* originated during the bubble years of the late '90s. It was (and still is, although like "Where's the beef?", it's not used much any more) a term meant to espouse the belief that everything moved faster on the Internet because getting information out with the Internet was so much faster than before.

But the underlying problem that we have been talking about all along remains: we are still asking the server to redraw entire screens for virtually every little user event and still asking the server to do the vast majority of the work of our application. We have evolved slightly from the earlier incarnations of the Web to the extent that client-side scripting is now available. So in fact, in contrast to how it was just a few years ago, not every user event requires a server call. Simple things like trivial data entry validations and such are now commonly performed on the client machine, independent of the server. Still, the fact remains that most events do require intervention by the server, whether or not that is ideal to do so. Also, where a user has scripting disabled, good design dictates that the site should "degrade" gracefully, which means the server again takes on most of the work anyway.

At this point I would like to introduce a term I am fond of that I may or may not have invented (I had never heard anyone use it before me, but I cannot imagine I was the first). The term is the *classic* Web. The classic Web to me means the paradigm where the server, for nearly every user event, redraws the entire screen. This is how webapps have been built for about 15 years now, since the Web first began to be known in a broad sense.

The webapps that have been built for many years now—indeed are still built today on a daily basis—have one big problem: they are by and large redrawing the entire screen each time some event occurs. They are intrinsically server-centric to a large degree. When the user does something, beyond some trivial things that can be processed client side such as mouse-over effects and such, the server must necessarily be involved. It has to do some processing, and then redraw what the user sees to incorporate the applicable updated data elements. This is, as I'm sure you have guessed, highly inefficient.

You may be asking yourself, "If we've been doing the classic Web thing for so long, and even still do it today, what's wrong with it?" We'll be discussing that very question next, but before we go further with that thought, let's go back briefly to our timeline. Do you see the pattern now? We started with centralized applications and complete redrawing of the screen in response to every single user action. Then we went to fat clients that largely ran locally and only updated the relevant portion of the screen. Then we went back to centralized applications, and also back to the central machine redrawing the entire screen.

So, what comes next? Well, simply put, the pendulum is in full swing right back the other direction, and with a ton of momentum to boot! Let's not jump to that quite yet though—let's first discuss why that shift is necessary in the first place.

What's So Wrong with the Classic Web?

In many ways, absolutely nothing! In fact, there is still a great deal of value to that way of designing webapps. The classic Web is great for largely linear application flows, and is also a wonderful medium for delivering information in an accessible way. It is easy for most people

to publish information and to even create rudimentary applications with basic user interactions. The classic Web is efficient, simple, ubiquitous, and accessible to most people. It is not, however, an ideal environment for developing complex applications. The fact that people have been able to do so to this point is a testament to the ingenuity of engineers rather than an endorsement of the Web as an application distribution medium!

It makes sense to differentiate now between a webapp and a web site, as summarized in Table 1-1.

Table 1-1. *Summary Comparison of Webapps vs. Web Sites*

Webapps	Web Sites
Designed with much greater user interaction in mind.	Very little user interaction aside from navigation.
Main purpose is to perform some function or functions, usually in real time, based on user inputs.	Main purpose is to deliver information, period.
Uses techniques that require a lot more of the clients accessing them in terms of client capabilities.	Tends to be created for the lowest common denominator.
Accessibility tends to take a back seat to functionality out of necessity and the simple fact that it's hard to do complex and yet accessible webapps.	Accessibility is usually considered and implemented to allow for the widest possible audience.
Tends to be more event-based and nonlinear.	Tends to be somewhat linear with a path the user is generally expected to follow with only minor deviations.

There are really two different purposes served by the Web at large. One is to deliver information. In this scenario, it is very important that the information be delivered in a manner that is readily accessible to the widest possible audience. This means not only people with disabilities who are using screen readers and such devices, but also those using more limited capability devices like cell phones, PocketPCs, and kiosk terminals. In such situations, there tends to be no user interactions aside from jumping from static document to static document, or at most very little interaction via simple fill-out forms. This mode of operation for the Web, if you will, can be classified as web sites.

The category of webapps on the other hand has a wholly different focus. Webapps are not concerned with simply presenting information, but in performing some function based on what the user does and what data the user provides. The user can be another automated system in many cases, but usually we are talking about real flesh-and-blood human beings. Webapps tend to be more complex and much more demanding of the clients that access them. In this case, "clients" refer to web browsers.

This does not have to be true. There are indeed some very complex webapps out there that do not require any more capability of clients than a web site does. While it clearly is not impossible to build complex applications in the "web site" mode, it is limiting and more difficult to do well in terms of user-friendliness, and it tends to require sacrifices in terms of capabilities or robustness of the capabilities provided.

This is the problem with the classic model: you generally have to design to the lowest common denominator, which severely limits what you can do.

Let's think a moment about what the lowest common denominator means in this context. Consider what you could and could not use to reach the absolute widest possible audience out there today. Here is a list of what comes to mind:

- *Client-side scripting*: Nope, you could not use this because many mobile devices do not yet have scripting support or are severely limited. This does not even consider those people on full-blown PCs who simply choose to disable scripting for security or other reasons.
- *Cascading Style Sheets (CSS)*: You could use it, but you would have to be very careful to use an older specification to ensure most browsers would render it properly—none of the fancier CSS 2.0 capabilities, for instance.
- *Frames*: No, frames are not universally supported, especially on many portable devices. Even when they are supported, you need to be careful because a frame is essentially like having another browser instance in terms of memory (and in some cases it very literally is another browser instance), and this can be a major factor in mobile devices.
- *Graphics*: Graphics can be tricky in terms of accessibility because they tend to convey more information than an `alt` attribute can. So, some of the meaning of the graphic can easily be lost for those with disabilities, no matter how vigilant you are to help them.
- *Newer HTML specs*: There are still many people out there using older browsers that may not even support HTML 4.01, so to be safe you will probably want to code to HTML 3.0. You will lose some capabilities obviously in doing so.

Probably the most important element here is the lack of client-side scripting. Without client-side scripting, so many possibilities are not available to you as a developer. Most important in the context of this book is the fact that you have virtually no choice but to have the server deal with every single user interaction. You may be able to get away with some meta-refreshes in frames in some cases, or perhaps other tricks of the trade, but frames too are on the list, so you might not even have that option!

You may be wondering, "What is the problem with the server rendering entire pages?" Certainly there are benefits, and the inherent security of being in complete control of the runtime state of the application (i.e., the user can't hack the code, at least not as easily) is a big one. Not having to incur the initial startup delay of downloading the code to the client is another. However, there are indeed some problems that in many cases overshadow the benefits. Perhaps the most obvious is the load on the server. Asking a server to do all this work on behalf of the client many times over across a number of simultaneous requests means that the server must be more robust and capable than it might otherwise need to be. This all translates to dollars and cents in the long run because you will have to purchase more server power to handle the load. Now, many people have the "just throw more hardware at it" mentality, and we are indeed in an age where that works most of the time. But that is much like saying that because we can throw bigger and bigger engines in cars to make them go faster, then that's exactly what we should always do when we need or want more speed. In fact, we can make cars go faster by making a smaller engine more efficient in design and execution, which in many ways is much more desirable, that is, if you like clean, fresh air to breathe!

Perhaps an even better metaphor would be to say it is like taking a midsized car and continually adding seats tied to it around the outside to allow for more people to ride "in" the car rather than trying to find a more efficient way for them to get where they are going. While this

duct-tape solution might work for a while, eventually someone is going to fall off and get crushed by the 18-wheeler driving behind us!

Another problem with the server-does-it-all approach is that of network traffic. Network technology continues to grow in leaps and bounds at a fantastic rate. Many of us now have broadband connections in our homes that we could not fully saturate if we tried (and I for one have tried!). However, that does not mean we should have applications that are sending far more information per request than they need to. We should still strive for thriftiness, should we not?

The other big problem is simply how the user perceives the application. When the server has to redraw the entire screen, it generally results in a longer wait time to see the results, not to mention the visual redrawing that many times occurs in webapps, flickering, and things of that nature. These are things users universally dislike in a big way. They also do not like losing everything they entered when something goes wrong, which is another common failing of the classic model.

At the end of the day, the classic model still works well on a small scale and for delivering mostly static information, but it doesn't scale very well, and it doesn't deal with the dynamic nature of the Web today nearly as well. In this context, "scale" refers to added functionality in the application, not simultaneous request-handling capability (although it is quite possible that is in play, too). If things do not work as smoothly, or if breakages result in too much lost, or if perceived speed is diminished, then the approach didn't scale well.

The classic model will continue to serve us well for some time to come in the realm of web sites, but in the realm of webapps, the realm you are likely interested in if you are reading this book, its demise is at hand, and its slayer is the hero of our tale: Ajax!

Enter Ajax

Ajax (see Figure 1-6 . . . now you'll always know what code and architectures would look like personified as a plucky super hero!) came to life, so to speak, at the hands of one Jesse James Garrett of Adaptive Path (`www.adaptivepath.com`). I am fighting my natural urge to make the obvious outlaw jokes here! Mr. Garrett wrote an essay in February 2005 (you can see it here: `www.adaptivepath.com/publications/essays/archives/000385.php`) in which he coined the term Ajax.

Figure 1-6. *Ajax to the rescue!*

In Figures 1-7 through 1-9, you can see some examples of Ajax-enabled applications. There are, of course, tons of examples from which to choose at this point, but these show diversity in what can be accomplished with Ajax.

Figure 1-7. *The PhotoShare application, one of the Ajax applications from my first book,* Practical Ajax Projects with Java Technology

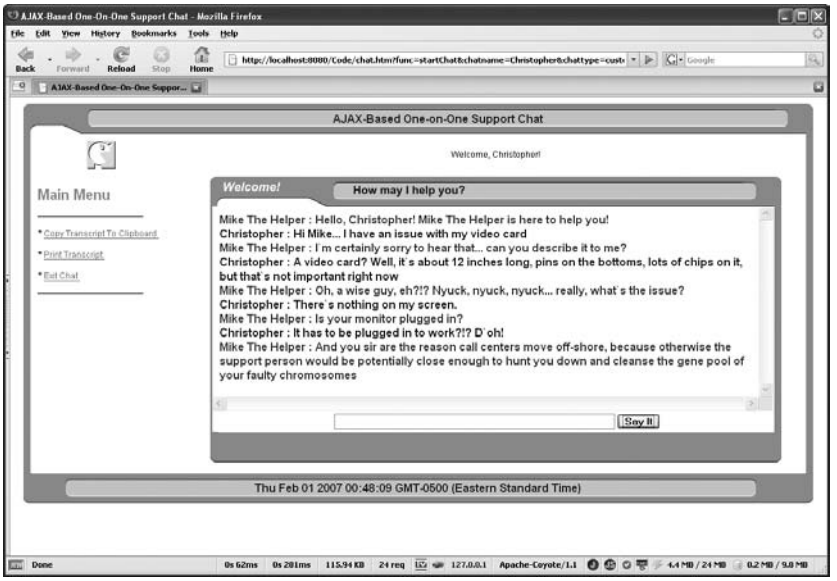

AJAX-Based One-On-One Support Chat - Mozilla Firefox
AJAX-Based One-on-One Support Chat
Welcome, Christopher!
Main Menu
Copy Transcript To Clipboard
Print Transcript
Exit Chat
Welcome!
How may I help you?
Mike The Helper : Hello, Christopher! Mike The Helper is here to help you!
Christopher : Hi Mike... I have an issue with my video card
Mike The Helper : I'm certainly sorry to hear that... can you describe it to me?
Christopher : A video card? Well, it's about 12 inches long, pins on the bottoms, lots of chips on it, but that's not important right now
Mike The Helper : Oh, a wise guy, eh?!? Nyuck, nyuck, nyuck... really, what's the issue?
Christopher : There's nothing on my screen.
Mike The Helper : Is your monitor plugged in?
Christopher : It has to be plugged in to work?!? D'oh!
Mike The Helper : And you sir are the reason call centers move off-shore, because otherwise the support person would be potentially close enough to hunt you down and cleanse the gene pool of your faulty chromosomes
Say It
Thu Feb 01 2007 00:48:09 GMT-0500 (Eastern Standard Time)
Done

Figure 1-8. *An Ajax-based chat application, this one from my second book,* Practical JavaScript, DOM Scripting, and Ajax Projects *(that's the good thing about writing subsequent books: you have source material for new ones!)*

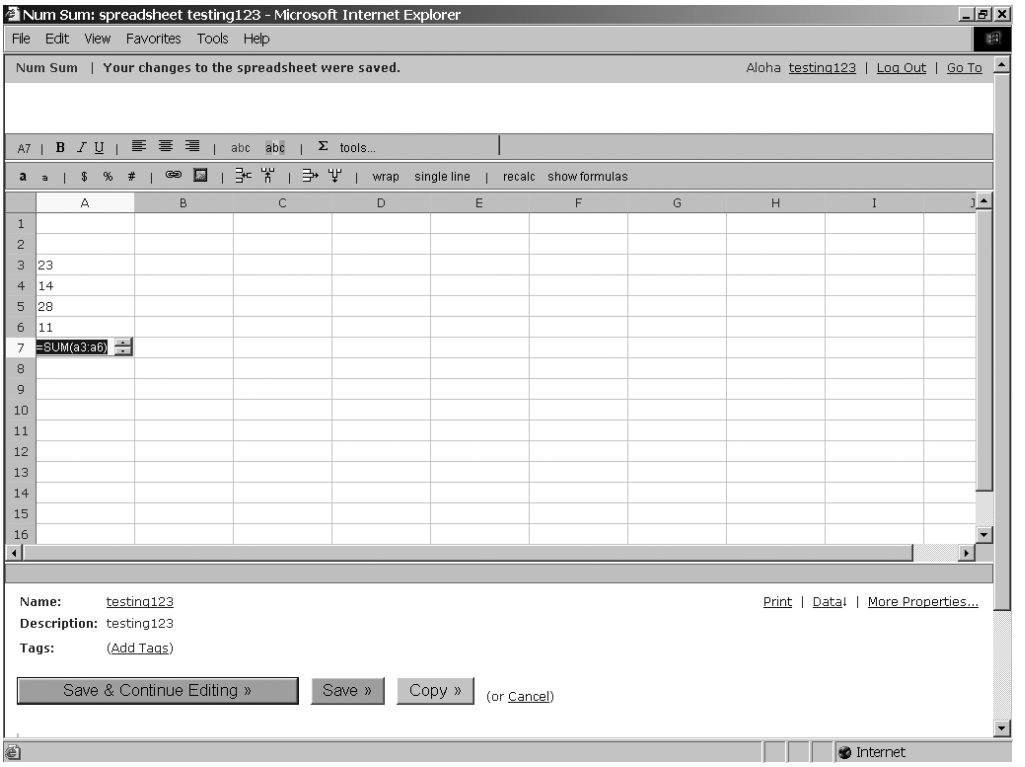


Figure 1-9. *Num Sum is an excellent Ajax-based spreadsheet application.*

Ajax, as I'd be willing to bet my dog you know already (well, not really, my wife and kids will kill me if I gave away the family dog, although my wallet would thank me), stands for Asynchronous JavaScript and XML. The interesting thing about Ajax, though, is that it doesn't have to be asynchronous (but virtually always is), doesn't have to involve JavaScript (but virtually always does), and doesn't need to use XML at all (and more and more frequently doesn't). In fact, one of the most famous Ajax examples, Google Suggest, doesn't pass back XML at all! The fact is that it doesn't even pass back data per se; it passes back JavaScript that contains data! (The data is essentially "wrapped" in JavaScript, which is then interpreted and executed upon return to the browser. It then writes out the list of drop-down results you see as you type.)

Ajax is, at its core, an exceedingly simple, and by no stretch of the imagination original, concept: it is not necessary to refresh the entire contents of a web page for each user interaction, or each *event*, if you will. When the user clicks a button, it is no longer necessary to ask the server to render an entirely new page, as is the case with the classic Web. Instead, you can define regions on the page to be updated, and have much more fine-grained control over user events as well. No longer are you limited to simply submitting a form or navigating to a new page when a link is clicked. You can now do something in direct response to a non-submit button being clicked, a key being pressed in a text box—in fact, to any event happening! The server is no longer completely responsible for rendering what the user sees; some of this logic is now performed in the user's browser. In fact, in a great many cases it is considerably better

to simply return a set of data and not a bunch of markup for the browser to display. As we traced along our admittedly rough history of application development, we saw that the classic model of web development is in a sense an aberration to the extent that we actually had it right before then!

Ajax is a return to that thinking. Notice I said "thinking." That should be a very big clue to you about what Ajax really is. It is not a specific technology, and it is not the myriad toolkits available for doing Ajax, and it is not the `XMLHttpRequest` object (don't worry if you've never seen that before, we'll get to it soon enough). It is a way of thinking, an approach to application development, a mindset.

The interesting thing about Ajax is that it is in no way, shape, or form new; only the term used to describe it is. I was reminded of this fact a while ago at the Philadelphia Java Users Group. A speaker by the name of Steve Banfield was talking about Ajax, and he said (paraphrasing from memory), "You can always tell someone who has actually done Ajax because they are pissed that it is all of a sudden popular." This could not be truer! I was one of those people doing Ajax years and years ago; I just never thought what I was doing was anything special and hence did not give it a "proper" name. Mr. Garrett holds that distinction.

I mentioned that I personally have been doing Ajax for a number of years, and that is true. What I did not say, however, is that I have been using XML or that I have been using the `XMLHttpRequest` object, or any of the Ajax toolkits out there. I've written a number of applications in the past that pulled tricks with hidden frames and returning data to them, then using that data to populate existing portions of the screen. This data was sometimes in the form of XML, other times not. The important point here is that the approach that is at the heart of Ajax is nothing new as it does not, contrary to its very own name, require any specific technologies (aside from client-side scripting, which is, with few exceptions, required of an Ajax or Ajax-like solution).

When you get into the Ajax frame of mind, which is what we are really talking about, you are no longer bound by the rules of the classic Web. You can now take back at least some of the power the fat clients offered, while still keeping the benefits of the Web in place. Those benefits begin, most importantly perhaps, with the ubiquity of the web browser.

SHAMELESS SELF-PROMOTION

If you're looking for a book on Ajax that touches on a number of the libraries out there available to help you get the job done, might I suggest my own *Practical Ajax Projects with Java Technology* (Apress, 2006)? It not only covers DWR, but also Dojo, the AjaxParts Taglib (APT) in Java Web Parts (JWP), and more. It does so in the same way as this book, that is, as a collection of full, working applications.

If you're looking for something that covers Ajax, but not as the focal point, then you might be interested in my book *Practical JavaScript, DOM Scripting, and Ajax Projects* (Apress, 2007). Whereas the first book is very much Java-centric, the second virtually never touches on the server side of things, squarely focusing on the client side and covering even more libraries: Dojo, Prototype, MooTools, and YUI, just to name a few.

Both of these books should be on your bookshelf because, aside from the fact that my son is addicted to video games and needs to feed his habit and because my daughter is a true princess by any definition and needs all the clothes and trimmings to go along with it, they're good references and learning materials besides (and hopefully there's some entertainment value there as well!).

Have you ever been at work and had to give a demo of some new fat-client app, for example, a Visual Basic app, that you ran on a machine you have never touched before? Ever have to do it in the boardroom in front of top company executives? Ever had that demo fail miserably because of some DLL conflict you couldn't possibly anticipate (see Figure 1-10)? You are a developer, so the answer to all of those questions is likely yes (unless you work in the public sector, and then it probably was not corporate executives, which I suppose means you may have run the risk of being lined up against a wall and shot for your "crimes," but either way, you get the point). If you have never done Windows development, you may not have had these experiences (yeah, right . . . if you believe it only happens on Windows, then I've got a big hunk of cheese to sell you . . . it's on display every evening, just look up in the sky and check it out). You will have to take my word for it when I say that such situations were, for a long time, much more common than any of us would have liked. With a web-based application, this is generally not a concern. Ensure the PC has the correct browser and version, and off you go 98 percent of the time.

Figure 1-10. *We've all been there: live demos and engineers do not mix!*

The other major benefit of a webapp is distribution. No longer do you need a 3-month shakedown period to ensure your new application does not conflict with the existing suite of corporate applications. An app running in a web browser, security issues aside, will not affect, or be affected by, any other application on the PC (and I am sure we all have war stories about exceptions to that, but they are just that: exceptions!).

Of course, you probably knew those benefits already, or you probably wouldn't be interested in web development in the first place, so we won't spend any more time on this.

Why Is Ajax a Paradigm Shift? On the Road to RIAs

Ajax does in fact represent a paradigm shift for some people (even most people, given what most webapps are today) because it can fundamentally change the way you develop a webapp. More important perhaps is that it represents a paradigm shift for the *user*, and in fact it is the user who will drive the adoption of Ajax. Believe me; you can no longer ignore Ajax as a tool in your toolbox.

Put a non-Ajax webapp in front of users, and then put that same app using Ajax techniques in front of them, and guess which one they are going to want to use all day nine times out of ten? The Ajax version! They will immediately see the increased responsiveness of the application and will notice that they no longer need to wait for a response from the server while they stare at a spinning browser logo wondering if anything is actually happening. They will see that the application alerts them on the fly of error conditions they would have to wait for the server to tell them about in the non-Ajax webapp. They will see functionality like type-ahead suggestions and instantly sortable tables and master-detail displays that update in real time—things that they would *not* see in a non-Ajax webapp. They will see maps that they can drag around like they can in the full-blown mapping applications they spent $80 on before. All of these things will be obvious advantages to the user. Users have become accustomed to the classic webapp model, but when confronted with something that harkens back to those fat-client days in terms of user-friendliness and responsiveness, there is almost an instantaneous realization that the Web as they knew it is dead, or at least should be!

If you think about many of the big technologies to come down the pike in recent years, it should occur to you that we technology folks rather than the users were driving many of them. Do you think a user ever asked for an Enterprise JavaBean (EJB)–based application? No, we just all thought it was a good idea (how wrong we were there!). What about web services? Remember when they were going to fundamentally change the way the world of application construction worked? Sure, we are using them today, but are they, by and large, much more than an interface between cooperating systems? Not usually. Whatever happened to Universal Description, Discovery, and Integration (UDDI) directories and giving an application the ability to find, dynamically link to, and use a registered service on the fly? How good did that sound? To us geeks it was the next coming, but it didn't even register with users.

Ajax is different, though. Users can see the benefits. They are very real and very tangible to them. In fact, we as technology people, especially those of us doing Java web development, may even recoil at Ajax at first because more is being done on the client, which is contrary to what we have been drilling into our brains all these years. After all, we all believe scriptlets in JavaServer Pages (JSPs) are bad, eschewing them in favor of custom tags. Users do not care about elegant architectures and separation of concerns and abstractions allowing for code reuse. Users just want to be able to drag the map around in Google Maps (see Figure 1-11) and have it happen in real time without waiting for the whole page to refresh like they do (or did anyway) when using Yahoo's mapping solution.

The difference is clear. They want it, and they want it now (stop snickering in your head, we're all adults here!).

Ajax is not the only new term floating around these days that essentially refers to the same thing. You may have also heard of Web 2.0 and RIAs. RIA is a term I particularly like, and I will discuss it in a bit.

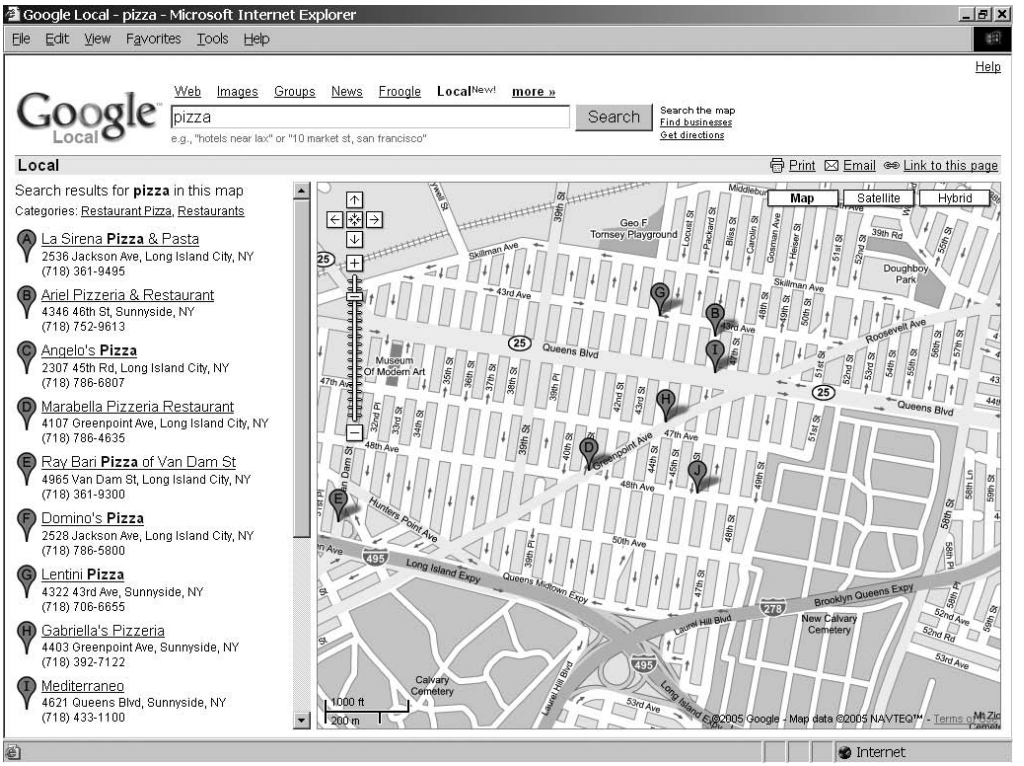


Figure 1-11. *Google Maps, a fantastic example of the power of Ajax*

WITHER WEB 2.0?

Web 2.0 is a term I'm not especially a fan of, partially because its meaning isn't really fixed precisely enough. For instance, if you take a cruise over to Wikipedia, you'll find this definition (assuming no one has changed it since I wrote this):

> *Web 2.0, a phrase coined by O'Reilly Media in 2003 and popularized by the first Web 2.0 conference in 2004, refers to a perceived second-generation of web-based communities and hosted services such as social networking sites, wikis, and folksonomies that facilitate collaboration and sharing between users. O'Reilly Media titled a series of conferences around the phrase, and it has since become widely adopted.*
>
> *Though the term suggests a new version of the Web, it does not refer to an update to World Wide Web technical specifications, but to changes in the ways systems developers have used the web platform. According to Tim O'Reilly, "Web 2.0 is the business revolution in the computer industry caused by the move to the Internet as platform, and an attempt to understand the rules for success on that new platform."*

> *Some technology experts, notably Tim Berners-Lee, have questioned whether one can use the term in a meaningful way, since many of the technology components of "Web 2.0" have existed since the beginnings of the World Wide Web.*
>
> Isn't it ironic that the man most directly responsible for the Internet in the first place, Tim Berners-Lee, basically says the same thing about the term? Sounds to me like someone just needed a term around which to base conferences, but that's more cynical a view than even **I** am comfortable with!
>
> The problem is that if you go somewhere else and look up the definition, you may well find something completely different. To some, for instance, Web 2.0 means the Web with all the Ajax goodness thrown in. To others, it means all sorts of transition effects, animations, and that sort of thing. To yet others, it means the ability of one system to connect with another via some form of web service. For some, it's a return to simplistic designs without tables using only CSS.
>
> In truth, all those things probably factor into the equation, but whichever meaning you like, Web 2.0 is a term that you'll hear a lot, and my suggestion is to try and glean its meaning not from any formal definition but rather from the context of the discussion you're involved in because it's likely the only context in which that particular definition of Web 2.0 will have any meaning!
>
> Oh yeah, and you should **definitely** put it on your resume somewhere. HR drones just **love** seeing it thrown around as if **you** are the only one with **THE** definition of it!

RIA stands for Rich Internet Application. Although there is no formal definition with which I am familiar, most people get the gist of its meaning without having to Google for it.

In short, the goal of an RIA is to create an application that is web based—that is, it runs in a web browser but looks, feels, and functions more like a typical fat-client application than a "typical" web site. Things like partial-page updates are taken for granted, and hence Ajax is always involved in RIAs (although what form of Ajax is involved can vary; indeed you may not find the `XMLHttpRequest` object, the prototypical Ajax solution, lurking about at all!). These types of applications are always more user-friendly and better received by the user community they service. In fact, your goal in building RIAs should be for users to say, "I didn't even know it was a webapp!"

Gmail (see Figure 1-12) is a good example of an RIA, although even it isn't perfect because while it has definite advantages over a typical web site, it still looks and feels very much like a web page, albeit one with lots of niceties mixed in. Microsoft's Hotmail is another good example (note that Hotmail is being supplanted by Microsoft Live Hotmail, which is the newer, more Web 2.0–ish version of Hotmail).

Yet another good example, shown in Figure 1-13, also comes courtesy of Google and is an online RSS feed reader.

You may have noticed that these examples of Ajax RIAs are from Google. That is not a coincidence. Google has done more to bring Ajax to the forefront of people's minds than anyone else. They were not the first to do it, or even the best necessarily, but they certainly have been some of the most visible examples and have really shown people what possibilities Ajax opens up.

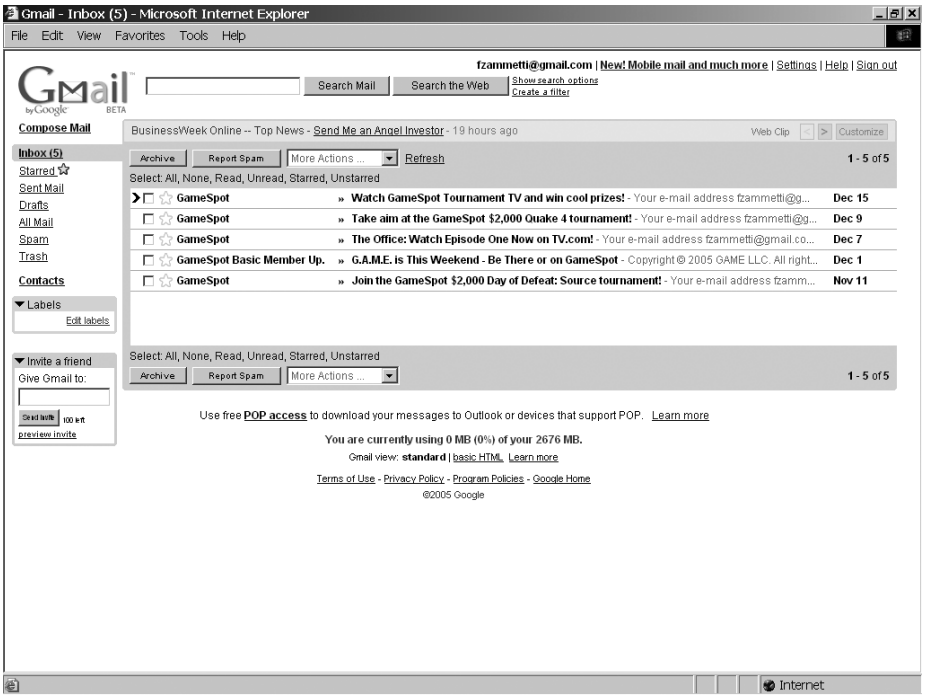

Gmail - Inbox (5) - Microsoft Internet Explorer
File Edit View Favorites Tools Help
fzammetti@gmail.com | New! Mobile mail and much more | Settings | Help | Sign out
Search Mail
Search the Web
Show search options
Create a filter
Compose Mail
Inbox (5)
Starred
Sent Mail
Drafts
All Mail
Spam
Trash
Contacts
Labels
Edit labels
Invite a friend
Give Gmail to:
Send Invite
100 left
preview invite
BusinessWeek Online -- Top News - Send Me an Angel Investor - 19 hours ago
Web Clip
Customize
Archive
Report Spam
More Actions ...
Refresh
1 - 5 of 5
Select: All, None, Read, Unread, Starred, Unstarred
GameSpot » Watch GameSpot Tournament TV and win cool prizes! - Your e-mail address fzammetti@g... Dec 15
GameSpot » Take aim at the GameSpot $2,000 Quake 4 tournament! - Your e-mail address fzammetti@g... Dec 9
GameSpot » The Office: Watch Episode One Now on TV.com! - Your e-mail address fzammetti@gmail.co... Dec 7
GameSpot Basic Member Up. » G.A.M.E. is This Weekend - Be There or on GameSpot - Copyright © 2005 GAME LLC. All right... Dec 1
GameSpot » Join the GameSpot $2,000 Day of Defeat: Source tournament! - Your e-mail address fzamm... Nov 11
Use free POP access to download your messages to Outlook or devices that support POP. Learn more
You are currently using 0 MB (0%) of your 2676 MB.
Gmail view: standard | basic HTML Learn more
Terms of Use - Privacy Policy - Program Policies - Google Home
©2005 Google
Internet

Figure 1-12. *Gmail, an Ajax webmail application from Google*

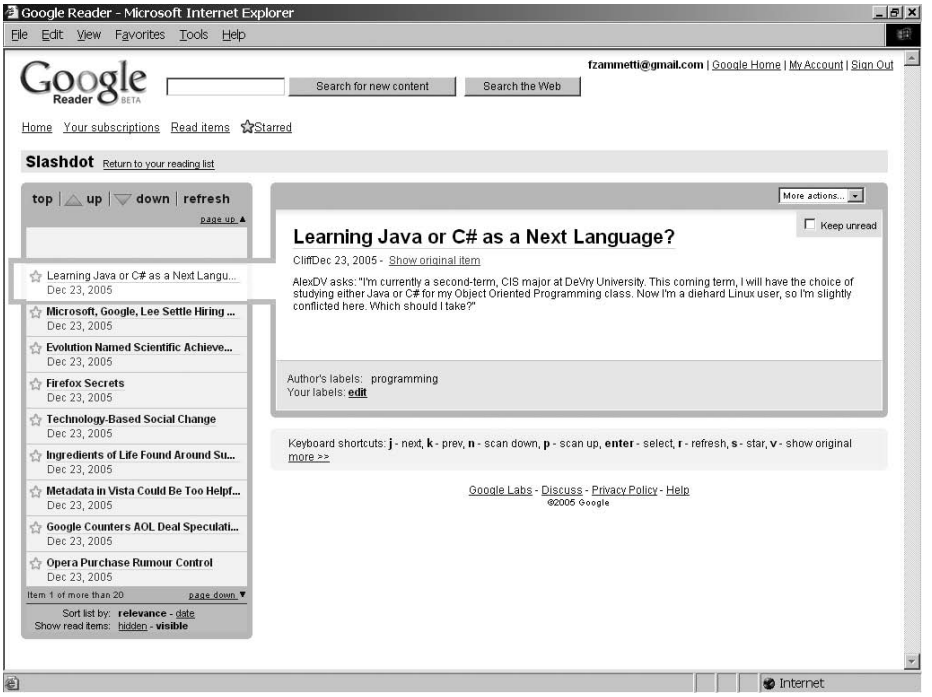

Google Reader - Microsoft Internet Explorer
File Edit View Favorites Tools Help
fzammetti@gmail.com | Google Home | My Account | Sign Out
Search for new content
Search the Web
Home Your subscriptions Read items Starred
Slashdot Return to your reading list
top | up | down | refresh
page up
Learning Java or C# as a Next Langu... Dec 23, 2005
Microsoft, Google, Lee Settle Hiring ... Dec 23, 2005
Evolution Named Scientific Achieve... Dec 23, 2005
Firefox Secrets Dec 23, 2005
Technology-Based Social Change Dec 23, 2005
Ingredients of Life Found Around Su... Dec 23, 2005
Metadata in Vista Could Be Too Helpf... Dec 23, 2005
Google Counters AOL Deal Speculati... Dec 23, 2005
Opera Purchase Rumour Control Dec 23, 2005
Item 1 of more than 20
page down
Sort list by: relevance - date
Show read items: hidden - visible
More actions...
Keep unread
Learning Java or C# as a Next Language?
CliffDec 23, 2005 - Show original item
AlexDV asks: "I'm currently a second-term, CIS major at DeVry University. This coming term, I will have the choice of studying either Java or C# for my Object Oriented Programming class. Now I'm a diehard Linux user, so I'm slightly conflicted here. Which should I take?"
Author's labels: programming
Your labels: edit
Keyboard shortcuts: j - next, k - prev, n - scan down, p - scan up, enter - select, r - refresh, s - star, v - show original
more >>
Google Labs - Discuss - Privacy Policy - Help
©2005 Google
Internet

Figure 1-13. *Google Reader is a rather good Ajax-enabled RSS reader*

The Flip Side of the Coin

Ajax sounds pretty darned good so far, huh? It is not all roses in Ajax land, however, and Ajax is not without its problems. Some of them are arguably only perceived problems, but others are concrete.

First and foremost, in my mind at least, is accessibility. You will lose at least some accessibility in your work by using Ajax because devices like screen readers are designed to read an entire page, and since you will no longer be sending back entire pages, screen readers will have trouble. My understanding is that some screen readers can deal with Ajax to some degree, largely depending on how Ajax is used. If the content is literally inserted into the Document Object Model (the DOM), using DOM manipulation methods, vs. just inserting content into a `<div>` using `innerHTML`, for example, that makes a big difference with regard to whether a screen reader will be able to handle it or not and alert the user of the change; using DOM methods is a little bit more supported at this point, although it's still not perfect. In any case, extreme caution should be used if you know people with disabilities are a target audience for your application, and you will seriously want to consider (and test!) whether Ajax will work in your situation. I am certain this problem will be addressed better as time goes on, but for now it is definitely a concern. Even still, there are some things you can do to improve accessibility:

- Put a note at the top of the page that says the page will be updated dynamically. This will give users the knowledge that they may need to periodically request a reread of the page from the screen reader to hear the dynamic updates.
- Depending on the nature of the Ajax you are using on a page, use `alert()` pop-ups when possible as these are read by a screen reader. This is a reasonable enough suggestion for things like Ajax-based form submission that will not be happening too frequently, but obviously if you have a timed, repeating Ajax event, this suggestion would not be a good one.
- Accessibility generally boils down to two main concerns: helping the vision-impaired and helping those with motor dysfunctions. Those with hearing problems tend to have fewer issues with web applications, although with more multimedia-rich applications coming online each day, this may be increasingly less true. Those with motor disorders will be concerned with things like keyboard shortcuts, since they tend to be easier to work with than mouse movements (and are generally easier than mouse movements for specialized devices to implement).
- Often overlooked is another kind of vision impairment: color blindness. Web developers usually do a good job of helping the blind, but they typically don't give as much attention to those who are color-blind. It is important to understand that color-blind people do not usually see the world in only black and white. Color blindness, or rather color deficiencies, is a failing of one of the three pigments that work in conjunction with the cone cells in your eyes. Each of the three pigments, as well as the cones, is sensitive to one of the three wavelengths of light: red, green, or blue. Normal eyesight, and therefore normal functioning of these pigments and cone cells, allows people to see very subtle differences in shades of the colors that can be made by mixing red, green, and blue. Someone with color blindness cannot distinguish these subtle shading differences as well as someone with normal color vision can, and sometimes cannot distinguish such differences at all. To someone with color blindness, a field of blue dots

with subtle red ones mixed in will appear as a field of dots all the same color, just as one example. A page that demonstrates the effects of color deficiencies to someone with normal vision can be found at `http://colorvisiontesting.com/what%20colorblind%20people%20see.htm`.

- Remember that it is not only those who have disabilities that have accessibility concerns, it can be folks with no impairments at all. For instance, you should try to use visual cues whenever possible. For instance, briefly highlighting items that have changed can be a big help. Some people call this the *Yellow Fade Effect*, whereby you highlight the changed item in yellow and then slowly fade it back to the nonhighlighted state. Of course, it does not have to be yellow, and it does not have to fade, but the underlying concept is the same: highlight changed information to provide a visual cue that something has happened. Remember that changes caused by Ajax can sometimes be very subtle, so anything you can do to help people notice them will be appreciated.

The term "Yellow Fade Effect" seems to have originated with a company called 37signals, as seen in this article: `www.37signals.com/svn/archives/000558.php`.

Another disadvantage of Ajax, many people feel, is added complexity. Many shops do not have in-house the client-side coding expertise Ajax requires (the use of toolkits that make it easier notwithstanding). The fact is, errors that originate client side are still, by and large, harder to track down than server-side problems, and Ajax does not make this any simpler. For example, View Source does not reflect changes made to the DOM (there are some tools available for Firefox that actually do allow this). Another issue is that Ajax applications will many times do away with some time-honored web concepts, most specifically back and forward buttons and bookmarking. Since there are no longer entire pages, but instead fragments of pages being returned, the browser cannot bookmark things in many cases. Moreover, the back and forward buttons cease to have the same meanings because they still refer to the last URL that was requested, and Ajax requests almost never are included (requests made through the `XMLHttpRequest` are not added to history, for example, because the URL generally does not change, especially when the method used is `POST`).

All of these disadvantages, except for perhaps accessibility to a somewhat lesser extent, have solutions, and we will see some of them later in the example apps. They do, however, represent differences in how webapps are developed for most developers, and they cause angst for many people. So they are things you should absolutely be aware of as you move forward with your Ajax work.

I'd say that's enough of theory, definitions, history, and philosophy behind Ajax and RIAs and all that. Let's go get our hands dirty with some actual code!

Let's Get to the Good Stuff: Our First Ajax Code, the Manual Way

This book aims to be different from most Ajax books in that it is based around the concept of giving you concrete examples to learn from, explaining them, explaining the decisions behind them (even the debatable ones), and letting you get your hands dirty with code. We are not going to spend a whole lot of time looking at UML diagrams, sequence diagrams, use case diagrams, and the like (although there will be some because, hey, we are code monkeys after all). You are more than welcome to pick up any of the fine UML books out there for that.

CODE MONKEY? DID YOU JUST CALL ME A CODE MONKEY?!?

Some people consider the term *code monkey* to be derogatory, but I never have. To me, a code monkey is someone who programs for a living and who enjoys writing code. I suppose by that definition you wouldn't even have to earn a living from programming, so long as you love hacking bits. Either way, it's all about the code!

By the way, just because someone is a code monkey doesn't mean they can't do architecture, and vice versa. If you like all of the facets of building software, if you like being up 'til all hours of the night trying to figure out why your custom-built double-linked list is corrupting elements when you modify them, if you like playing with that new open source library for no other reason than you're curious, if you like the feeling you get from seeing a working application come spewing out the back end of a development cycle (even if it's some otherwise dull business application), then you're a code monkey, plain and simple, and you should never take offense to being called that name.

Of course, some people **do** in fact mean it in a derogatory way, and you'll know who they are, in which case you should be a good little code monkey and throw feces at them (Frank Zammetti and Apress cannot be held liable if you actually follow this advice!).

Oh yes, and no discussion of the term code monkey would be complete with referencing the fantastic parody song "Code Monkey" by one Jonathan Coulton. His web site is here: `www.jonathancoulton.com`. Sadly, at the time I wrote this, it appeared to be having problems. Hopefully they are just temporary, but in any case, if you like Weird Al–style funny songs and want to hear a good one about us code monkeys, of which I am proudly one, try that site, and failing that, spend a few minutes with Google trying to find it (I don't even think the RIAA, which is the Recording Industry Association of America, the American organization famous for suing grandmothers, children, and even dead people for illegally downloading pirated music, will have a problem with it, but you never can tell with those people, so I'm **not** suggesting any file-sharing networks you might try!).

With that in mind, I am not going to waste any more time telling you what Ajax is, why it is the greatest thing since sliced bread, where the name came from, or any of that. Instead, we are going to jump right into some code!

This first example is somewhat unique in that it doesn't require Java. In fact, it does not require a server at all. Rest assured that all the other examples in this book do, just not this one. But we want to cover a simple Ajax app without server interaction first, just to get some of the basics covered, so here goes (see Listing 1-1).

Listing 1-1. *Our First Real Ajax Application! (This Is the File* index.htm*.)*

```
<html>

  <head>

    <title>Simple Non-Server AJAX Example</title>

    <script>

      // This is a reference to an XMLHttpRequest object.
      xhr = null;

      // This function is called any time a selection is made in the first
      // <select> element.
      function updateCharacters() {

        var selectedShow = document.getElementById("selShow").value;
        if (selectedShow == "") {
          document.getElementById("divCharacters").innerHTML = "";
          return;
        }

        // Instantiate an XMLHttpRequest object.
        if (window.XMLHttpRequest) {
          // Non-IE.
          xhr = new XMLHttpRequest();
        } else {
          // IE.
          xhr = new ActiveXObject("Microsoft.XMLHTTP");
        }
        xhr.onreadystatechange = callbackHandler;
        url = selectedShow + ".htm";
        xhr.open("post", url, true);
        xhr.send(null);

      }

      // This is the function that will repeatedly be called by our
      // XMLHttpRequest object during the life cycle of the request.
      function callbackHandler() {

        if (xhr.readyState == 4) {
          document.getElementById("divCharacters").innerHTML = xhr.responseText;
        }

      }
```

```
    </script>

  </head>

  <body>

    Our first simple AJAX example
    <br><br>

    Make a selection here:
    <br>
    <select onChange="updateCharacters();" id="selShow">
      <option value=""></option>
      <option value="b5">Babylon 5</option>
      <option value="bsg">Battlestar Galactica</option>
      <option value="sg1">Stargate SG-1</option>
      <option value="sttng">Star Trek The Next Generation</option>
    </select>

    <br><br>

    In response, a list of characters will appear here:
    <br>
    <div id="divCharacters">
      <select>
      </select>
    </div>

  </body>

</html>
```

Figure 1-14 shows what it looks like on the screen (don't expect much here, folks!).

As you can see, there is no content in the second drop-down menu initially. This will be dynamically populated once a selection is made in the first, as shown in Figure 1-15.

Figure 1-15 shows that when a selection is made in the first drop-down menu, the contents of the second are dynamically updated. In this case we see characters from the greatest television show ever, *Babylon 5* (don't bother arguing, you know I'm right! And besides, you'll get your chance to put in your favorites later!). Now let's see how this "magic" is accomplished.

Listing 1-1 shows the first page of our simple Ajax example, which performs a fairly typical Ajax-type function: populate one `<select>` box based on the selection made in another. This comes up all the time in web development, and the "classic" way of doing it is to submit a form, whether by virtue of a button the user has to click or by a JavaScript event handler, to the server and let it render the page anew with the updated contents for the second `<select>`. With Ajax, none of that is necessary.

Simple Non-Server AJAX Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Our first simple AJAX example
Make a selection here:
In response, a list of characters will appear here:
Done
My Computer

Figure 1-14. *Note that there is no content in the second drop-down menu because nothing has yet been selected in the first.*

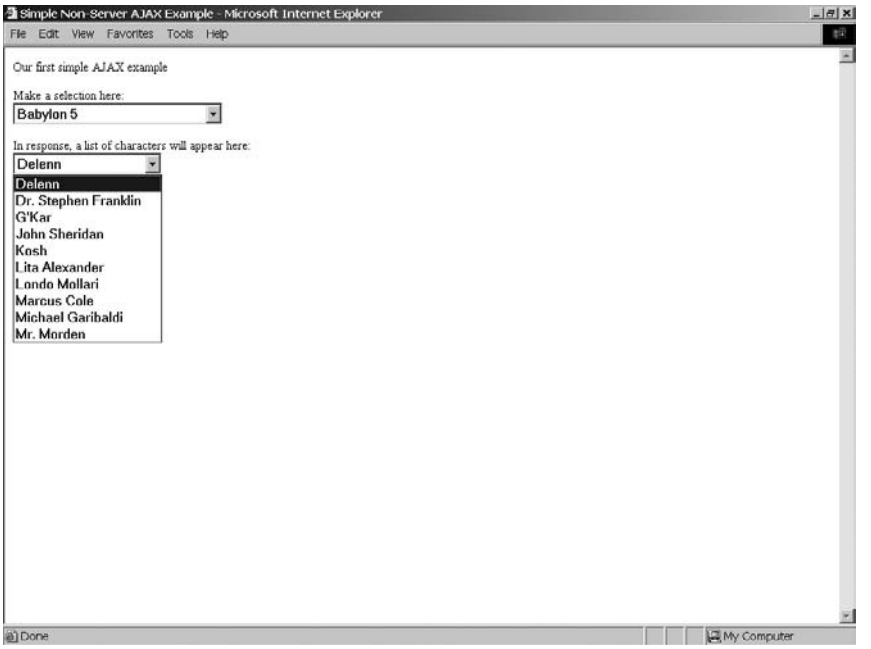


Figure 1-15. *A selection has been made in the first drop-down menu, and the contents of the second have been dynamically created from what was returned by the "server."*

A Quick Postmortem

Let's walk through the code and see what is going on. Note that this is not meant to be a robust, production-quality piece of code. It is meant to give you an understanding of basic Ajax techniques, nothing more. There is no need to write in with all the flaws you find!

First things first: the markup itself. In our `<body>` we have little more than some text and two `<select>` elements. Notice that they are not part of a `<form>`. You will find that forms tend to have less meaning in the world of Ajax. You will many times begin to treat all your form UI elements as top-level objects along with all the other elements on your page (in the `<body>` anyway).

Let's look at the first `<select>` element. This `<select>` element is given the ID `selShow`. This becomes a node in the DOM of the page. DOM, if you are unfamiliar with the term, is nothing more than a tree structure where each of the elements on your page can be found. In this case, we have a branch on our tree that is our `<select>` element, and we are naming it `selShow` so we can easily get at it later.

```
<select onChange="updateCharacters();" id="selShow">
  <option value=""></option>
  <option value="b5">Babylon 5</option>
  <option value="bsg">Battlestar Galactica</option>
  <option value="sg1">Stargate SG-1</option>
  <option value="sttng">Star Trek The Next Generation</option>
</select>
```

You will notice the JavaScript event handler attached to this element. Any time the value of the `<select>` changes, we will be calling the JavaScript function named `updateCharacters()`. This is where all the "magic" will happen. The rest of the element is nothing unusual. I have simply created an `<option>` for some of my favorite shows. After that we find another `<select>` element . . . sort of:

```
<div id="divCharacters">
  <select>
  </select>
</div>
```

It is indeed an empty `<select>` element, but wrapped in a `<div>`. You will find that probably the most commonly performed Ajax function is to replace the contents of some `<div>`. That is exactly what we will be doing here. In this case, what will be returned by the "server" (more on that in a minute) is the markup for our `<select>` element, complete with `<option>`'s listing characters from the selected show. So, when you make a show selection, the list of characters will be appropriately populated, and in true Ajax form, the whole page will not be redrawn, but only the portion that has changed—the second `<select>` element in this case (or more precisely, the `<div>` that wraps it) will be.

Let's quickly look at our mock server. Each of the shows in the first `<select>` has its own HTML file that in essence represents a server process. You have to take a leap of faith here and pretend a server was rendering the response that is those HTML pages. They all look virtually the same, so I will only show one as an example (see Listing 1-2).

Listing 1-2. *Sample Response Listing Characters from the Greatest Show Ever,* Babylon 5*!*

```
<select>
  <option>Delenn</option>
  <option>Dr. Stephen Franklin</option>
  <option>G'Kar</option>
  <option>John Sheridan</option>
  <option>Kosh</option>
  <option>Lita Alexander</option>
  <option>Londo Mollari</option>
  <option>Marcus Cole</option>
  <option>Michael Garibaldi</option>
  <option>Mr. Morden</option>
</select>
```

As expected, it really is nothing but the markup for our second `<select>` element.

Hey, I Thought This Was Ajax?!?

So, now we come to the part that does all the work here, our JavaScript function(s). First is the `updateCharacters()` function, shown here:

```
// This function is called any time a selection is made in the first
// <select> element.
function updateCharacters() {

        var selectedShow = document.getElementById("selShow").value;
        if (selectedShow == "") {
          document.getElementById("divCharacters").innerHTML = "";
          return;
        }

  // Instantiate an XMLHttpRequest object.
  if (window.XMLHttpRequest) {
    // Non-IE.
    xhr = new XMLHttpRequest();
  } else {
    // IE.
    xhr = new ActiveXObject("Microsoft.XMLHTTP");
  }
  xhr.onreadystatechange = callbackHandler;
  url = selectedShow + ".htm";
  xhr.open("post", url, true);
  xhr.send(null);

}
```

If you were to do manual Ajax on a regular basis, this basic code would very soon be imprinted on the insides of your eyelids (fortunately for you, this book is about DWR, so you

won't have to deal with these details manually, but it's still good to get an idea what's going on under the covers, and that's exactly what this is). This is the basic prototypical Ajax function, and you'd find similar code in an Ajax application, or in any library providing Ajax functionality. Let's tear it apart, shall we?

First, the function deals with the case where the user selects the blank option from the drop-down menu. In that case, we want to remove any second drop-down that may be present, so it's a simple matter of writing a blank string to the `innerHTML` property of the target `<div>`.

The first thing we need to add to our Ajax call, as one would expect, is an `XMLHttpRequest` object. This object, a creation of Microsoft (believe it or not!), is nothing more than a proxy to a socket. It has a few (very few) methods and properties, but that is one of the benefits: it really is a very simple beast.

Notice the branching logic here. It turns out that getting an instance of the `XMLHttpRequest` object is different in Internet Explorer than in any other browser (although it's worth noting that in IE7, an `XMLHttpRequest` object **is** now available, which means the same instantiation code can be used across browsers! However, to support the widest possible audience, if you have to code manual Ajax, you will want to stick with this type of branched code, which still works in IE7 as well). Now, before you get your knickers in a knot and get your anti-Microsoft ire up, note that Microsoft invented this object, and it was the rest of the world that followed. So, while it would be nice if Microsoft updated its API to match everyone else's, it isn't Microsoft's fault we need this branching logic! The others could just as easily have duplicated what Microsoft did exactly too, so let's not throw stones here—we're all in glass houses on this one! (Well, that's not **technically** true because Microsoft's implementation uses ActiveX, which means only Windows-based browsers could really implement it the same way. On the other hand, we could envision a very simple emulation of ActiveX to the extent that the browser could recognize this particular instantiation method and spawn the native object, as it would do normally in the absence of such emulation, so it still really does hold true.)

This is probably a good time to point out that `XMLHttpRequest` is pretty much a de facto standard at this point. It is also being made a true W3C standard, but for now it is not. It is safe to assume that any "modern" browser—that is, a desktop web browser that is no more than a few versions old—will have this object available. More limited devices, such as PocketPCs, cell phones, and the like, will many times not have it, but by and large it is a pretty ubiquitous little piece of code.

Continuing on in our code review . . . once we have an `XMLHttpRequest` object instance, we assign the reference to it to the variable `xhr` in the global page scope. Think about this for just a minute; what happens if more than one `onChange` event fires at close to the same time? Essentially, the first will be lost because a new `XMLHttpRequest` object is spawned, and `xhr` will point to it. Worse still, because of the asynchronous nature of `XMLHttpRequest`, a situation can arise where the callback function for the first request is executing when the reference is `null`, which means that callback would throw errors due to trying to reference a `null` object. If that was not bad enough, this will be the case only in some browsers, but not all (although my research indicates most would throw errors), so it might not even be a consistent problem.

Remember, I said this was not robust, production-quality code! This is a good example of why. That being said, it is actually many times perfectly acceptable to simply instantiate a new instance and start a new request. Think about a fat client that you use frequently. Can you spot instances where you can kick off an event that in essence cancels a previous event that was in the process of executing? For example, in your web browser, can you click the Home button

while a page is loading, thereby causing the page load to be prematurely ended and the new page to begin loading? Yes you can, and that is what in essence happens by starting a new Ajax request using the same reference variable. It is not an unusual way for an application to work, and sometimes is downright desirable. That being said, though, you do need to keep it in mind and be sure it is how you want and need things to work.

The next step we need to accomplish is telling the `XMLHttpRequest` instance what callback handler function to use. An Ajax request has a well-defined and specific life cycle, just like any HTTP request (and keep in mind, that's all an Ajax request is at the end of the day: a plain ole HTTP request!). This cycle is defined as the transitions between ready states (hence the property name, `onreadystatechange`). At specific intervals in this life cycle, the JavaScript function you name as the callback handler will be called. For instance, when the request begins, your function will be called. As the request is chunked back to the browser, in most browsers at least (IE being the unfortunate exception), you will get a call for each chunk returned (think about those cool status bars you can finally do with no complex queuing and callback code on the server!). Most important for us in this case, the function will be called when the request completes. We will see this function in just a moment.

The next step is probably pretty obvious: we have to tell the object what URL we want to call. We do this by calling the `open()` method of the object. This method takes three parameters: the HTTP method to perform, the URL to contact, and whether we want the call to be performed asynchronously (true) or not (false). Because this is a simple example, each television show gets its own HTML file pretending to be the server. The name of the HTML file is simply the value from the `<select>` element with `.htm` appended to the end. So, for each selection the user makes, a different URL is called. This is obviously not how a real solution would work—the real thing would likely call the same URL with some sort of parameter to specify the selected show—but some sacrifices were necessary to keep the example both simple and not needing anything on the server side of things.

The HTTP method can be any of the standard HTTP methods: `GET`, `POST`, `HEAD`, etc. Ninety-eight percent of the time you will likely be passing `GET` or `POST`. The URL is self-explanatory, except for one detail: if you are doing a `GET`, you must construct the query string yourself and append it to the URL. That is one of the drawbacks of `XMLHttpRequest`: you take full responsibility for marshaling and unmarshaling data sent and received. Remember, it is in essence just a very thin wrapper around a socket. This is where any of the numerous Ajax toolkits can come in quite handy, but we will talk about that later.

Once we have the callback registered with the object and we have told it what we're going to connect to and how, we simply call the `send()` method. In this case, we are not actually sending anything, so we pass `null`. One thing to be aware of is that, at least when I tested it, you can call `send()` with no arguments in IE and it will work, but in Firefox it will not. `null` works in both, though, so `null` it is.

Of course, if you actually had some content to send, you would do so here. You can pass a string of data into this method, and the data will be sent in the body of the HTTP request. Many times you will want to send actual parameters, and you do so by constructing essentially a query string in the typical form `var1=val1&var1=val1` and so forth, but without the leading question mark. Alternatively, you can pass in an XML DOM object, and it will be serialized to a string and sent. Lastly, you could send any arbitrary data you want. If a comma-separated list does the trick, you can send that. Anything other than a parameter string will require you to deal with it; the parameter string will result in request parameters as expected.

So far we've described how a request is sent. It is pretty trivial, right? Well, the next part is what can be even more trivial, or much more complex. In our example, it is the former. I am referring to the callback handler function, which is as follows:

```
// This is the function that will repeatedly be called by our
// XMLHttpRequest object during the life cycle of the request.
function callbackHandler() {
  if (xhr.readyState == 4) {
    document.getElementById("divCharacters").innerHTML = xhr.responseText;
  }
}
```

Our callback handler function this time around does very little. First, it checks the `readystate` of the `XMLHttpRequest` object. Remember I said this callback will be called multiple times during the life cycle of the request? Well, the `readystate` code you will see will vary with each life cycle event. The full list of codes (and there is only a handful) can be found with about five seconds of not-too-strenuous effort with Google, but for the purposes of this example we are only interested in code 4, which indicates the request has completed. Notice that I didn't say completed *successfully*! Regardless of the response from the server, the `readystate` will be 4. Since this is a simple example, we don't care what the server returns. If an HTTP 404 error (page not found) is received, we don't care in this case. If an HTTP 500 error (server processing error) is received, we still do not care. The function will do its thing in any of these cases. I repeat my refrain: this is not an industrial-strength example!

When the callback is called as a result of the request completing, we simply set the `innerHTML` property of the `<div>` on the page with the ID `divCharacters` to the text that was returned. In this case, the text returned is the markup for the populated `<select>`, and the end result is the second `<select>` is populated by characters from the selected show.

Now, that wasn't so bad, was it?

For a fun little exercise, and just to convince yourself of what is really going on, I suggest adding one or two of your own favorite shows in the first `<select>`, and creating the appropriately named HTML file to render the markup for the second `<select>`.

Cutting IN the Middle Man: Ajax Libraries to Ease Our Pain

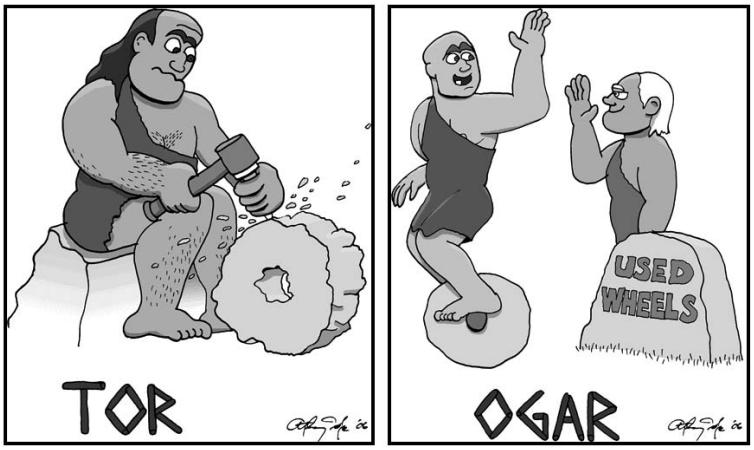
At this point you are probably thinking, "OK, Ajax is no big deal," and you would be right. At least in its more basic form, there is not much code to it, and it is pretty easy to follow.

However, if this is your first exposure to JavaScript and client-side development in general, it may seem like a lot of work. Or, if you intend to be doing more complex Ajax work, you may not want to be hand-coding handlers and functions all over the place. It is in these cases where a good Ajax library can come into play.

As you might imagine with all the hype surrounding Ajax recently, there are a ton of libraries and toolkits to choose from. Some are fairly blunt, general-purpose tools that make doing Ajax only a bit simpler, whereas others are rather robust libraries that seek to fill all your JavaScript and Ajax needs. Some provide powerful and fancy GUI widgets that use Ajax behind the scenes; others leave the widget building to you, but make it a lot easier.

Ajax libraries, JavaScript libraries in general really, have truly grown in leaps and bounds over the past two to three years. It used to be that you could spend a few hours scouring the web looking for a particular piece of code, and you indeed eventually found it. Often though, you might have to, ahem, appropriate it from some web site (this still happens of course). Many times, you could, and can, find what you need on one of a handful of *script sites* that are there expressly to supply developers with JavaScript snippets for their own use.

However, larger libraries that provide all sorts of bells and whistles, as exist in the big-brother world of Java, C++, PHP, and other languages, are a more recent development in the world of JavaScript. In many ways, we are now in a golden age in this regard, and you will find almost more options than you might want! So **use those libraries**, lest you wind up like Tor!



One thing that most of these modern-day libraries, at least the more well-known ones, have in common is that their quality is light-years beyond what used to be available. More importantly to you, each and every one of them will likely make your life (in terms of writing code anyway) considerably easier. There's usually no sense in reinventing the wheel after all. If you are doing Ajax, unless you need absolute control over every detail of things, I can't think of a good reason not to use a library for it. If you know your UI design requires some more advanced widgets that the browser doesn't natively provide, these libraries can absolutely be worth their weight in gold.

At the end of the day, though, keep in mind that "naked" Ajax, as seen in the example from this chapter, is also extremely common and has many of its own benefits, some of which are control over what is going on at the most fundamental level and more opportunity to tune for performance and overall robustness. It is also never a bad idea to know exactly what is going on in your own applications! If you are comfortable with JavaScript and client-side development in general, you may never need or want to touch any of these toolkits (I personally lean that way), but if you are just coming to the client-side party, these libraries can indeed make your life a lot easier.

JUST TO NAME A FEW . . .

There truly is a wealth of Ajax libraries out there, and of JavaScript libraries that have Ajax as one component. I saw a recent survey that listed something on the order of 236 choices at the moment, and that doesn't even count the ones that are specific to Java or .NET or PHP or what have you! As a service to you, dear reader, I'd like to list just a handful that I personally consider the cream of the crop. This isn't to shortchange any others because it's frankly likely I simply am not aware of some other great options, but these are ones I've personally used and think rather highly of (it's also true that most of these tend to top most peoples' lists).

- *Dojo* (`http://dojotoolkit.org`)

  Dojo is very much a jack-of-all-trades, covering a very broad range of topics, Ajax being just one of them. What Dojo is probably most famous for is its widget system and its widgets. It provides some very snazzy UI goodness, things like fisheye lists (think the dock bar on Macintosh systems) and more. Dojo is still a developing library, so expect some growing pains as you use it, but it gets a lot of attention and you'll quickly see why if you take a look.

- *MooTools* (`http://mootools.net`)

  MooTools is another of those "cover all the bases" libraries. One of its most interesting features is actually its web site! It allows you to build a custom version of it containing just the component you want. It deals with any dependencies that can arise so your build will always work. Aside from that, it has some great components including Ajax, of course, as well as effects and transitions and basic language enhancements.

- *Prototype* (`www.prototypejs.org`)

  Prototype serves as the foundation for many other famous libraries, and for good reason. It is small, relatively simple, and extremely useful. You can view it as an extension to JavaScript itself for the most part. You won't find some of the more fancy features of other libraries like widgets and effects, but it will enhance the language for you in ways you'll be very thankful for.

- *YUI* (`http://developer.yahoo.com/yui`)

  The Yahoo! User Interface library, while by and large about user interface widgets, also provides a good supply of utilities outside those widgets. YUI is one of the simpler and clearly written (and documented!) libraries out there. It won't wow people as much as something like Dojo will, but it makes up for it in spades in being very easy to use and very well supported.

- *script.aculo.us* (`http://script.aculo.us`)

  script.aculo.us, aside from being bizarrely named, is, to quote its tag line, "about the interface, baby!" Indeed it is. If you need visual effects, transitions, and so forth, this is the one library you'll want to check out. To see it in action, you can visit one of the more popular sites on the web: Digg (`www.digg.com`). Digg uses script.aculo.us to do all sorts of nifty little things along the way, like fading elements in and out when you post.

Alternatives to Ajax

No discussion of Ajax would be complete without pointing out that it is not the only game in town. There are other ways to accomplish the goals of building RIAs than using Ajax.

Have you ever heard of Flash? Of course you have! Who hasn't been annoyed by all those animations all over some of the most popular web sites around? Flash can be very annoying at times, but when used properly, it can provide what Ajax provides.

One good example is Mappr (`www.mappr.com`), which is a Flash-based application for exploring places based on pictures people take and submit for them (see Figure 1-16). This is an example of how Flash, used properly, can truly enhance the web users' experience.

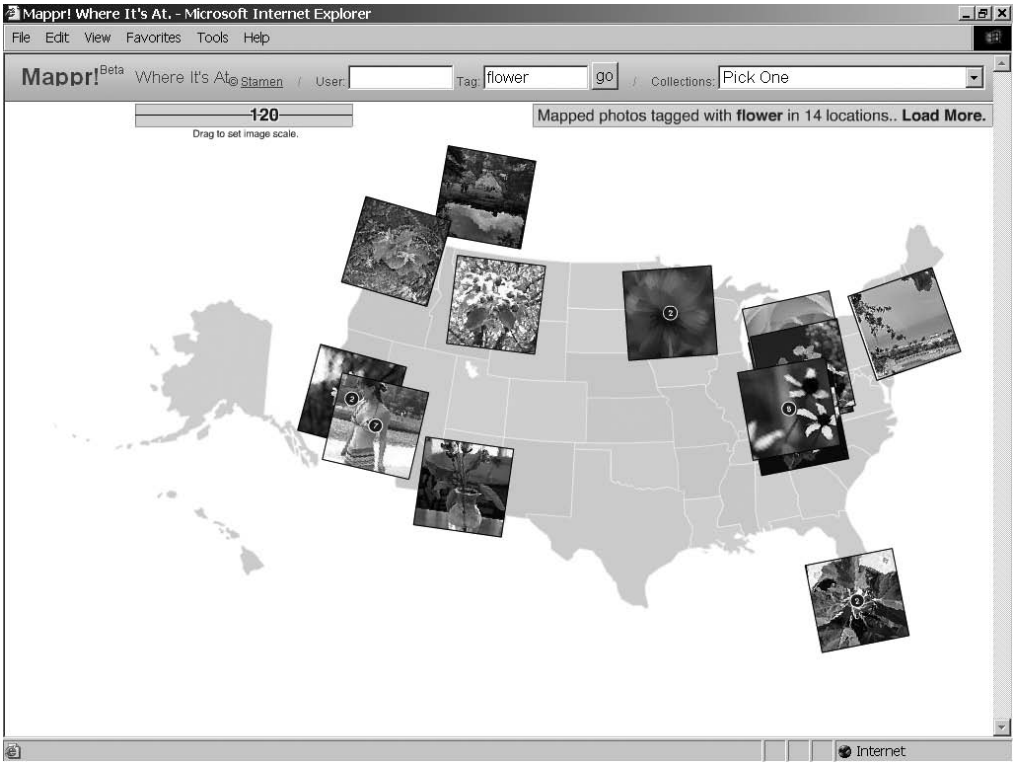


Figure 1-16. *Mappr, an excellent use of Flash*

Flash is a more "proper" development environment as well, complete with its own highly evolved integrated development environment (IDE), debugging facilities, and all that good stuff. It is also a tool that graphic designers can readily use to create some truly awe-inspiring multimedia presentations.

Flash is also fairly ubiquitous at this point. Virtually all browsers come with a Flash player bundled, or one can be added on with virtually no difficulty. Flash does, however, require a browser plug-in, whereas Ajax doesn't, and that is one advantage Ajax has. On the other hand, there are versions of Flash available for many mobile devices like PocketPCs and some cell phones, even those where `XMLHttpRequest` may not yet be available, so that's a plus for Flash.

Something of an extension to Flash is the relatively new product called Flex from Adobe. Flex is basically a markup-based UI toolkit built on top of Flash. What this means is you can code an application much like you do an HTML page, using special tags that represent UI elements such as tabbed dialogs, buttons, combo boxes, data grids and more, as well as script-based code written in Adobe's ActionScript language (which is fairly similar to JavaScript in most regards, certainly syntactically for the most part). The neat thing is that you compile a Flex page (which is a file with an extension of `.mxml`) into a Flash SWF file, which then runs in a Flash browser plug-in (or stand-alone Flash player), typically launched from an HTML page (just like any Flash movie embedded in a web page, but in this case the web page is literally nothing but a container and launcher for the Flash file). What this provides is a very rich UI experience, as well as providing more system-specific services such as the ability to write to the local file system, the ability to make HTTP requests to servers, the ability to make web service–like requests to a suitable server, and much more. Flex is starting to gain a lot of momentum among developers because it gives you the ability to present an extremely rich user interface in your applications while having most of the benefits of HTML and JavaScript-based applications, including cross-platform support and no deployment woes, not to mention saving you having to deal with the sometimes sticky details of Ajax, concurrent user events, etc.

In Figure 1-17, you can see an example of a Flex application. This is one of the sample applications that comes packaged with the Flex SDK, which is a free download. That's right; you can get started with Flex today if you want, at absolutely no cost! The SDK is free, as is of course the Flash runtime. There is, however, a commercial Flex IDE that Adobe sells as well, which you may want to look at if you decide to work with Flex on a regular basis. As you might imagine, this example has a much greater "wow" factor if you actually play with it, so I very much recommend having a look if you think Flex might be something you are interested in. It showcases quite a lot of, well, **Flash**, that is, various animations and effects, as you navigate the application.

All in all, Flex is, in my opinion (and in the growing opinion of many) a very viable alternative to Ajax-based applications, and may even represent the single best alternative available today.

What are the negatives about Flash, and by extension Flex? First, it has a bit of a bad reputation because it is so easy to abuse. Many people choose to not install it or block all Flash content from their browser. Second, in the case of Flash, it is not completely "typical" programming; it is a little more like creating a presentation in the sense that you have to deal with the concept of a "timeline," so there can be a bit of a learning curve for "traditional" developers. This is not true of Flex, which is very much like creating typical web applications. However, keep in mind that developers will still have a learning curve as they will need to get up to speed on the UI markup language, as well as ActionScript. Third, it can be somewhat heavyweight for users not on broadband connections, and even for those *with* broadband connections it can sometimes take longer than is ideal to start up. Lastly, Flash and Flex are owned and developed by Adobe, and the Flash runtime is not open sourced, although Flex recently was open sourced. In my opinion, it should still be viewed as something where vendor lock-in could become an issue, although it's certainly not any worse than many other technologies we all use, and better than some.

All in all, as alternatives go, Flash, and Flex more specifically, are good things to consider as you decide how to develop your RIAs. There are certainly pluses and minuses down each path.

Figure 1-17. *The Flex store example application*

While not, strictly speaking, an "alternative" to Ajax, I feel that Google Web Toolkit, or GWT, should be discussed in this section as well. GWT is an interesting product because it allows you to build web applications in pure Java. That's right, no writing HTML or JavaScript, no dealing with various browser quirks and cross-browser concerns. Just write your code very much like you would a GUI application in Java and let GWT handle the rest. Just to prove it's not just a neat subject for a thesis paper, note that Google Maps and Gmail are both built using GWT.

Now, some among you, and frankly me included, may feel that this is a little too unique. Speaking for myself, I've never been a huge fan of the way GUIs are written in Java, and to write a webapp like that doesn't really fill me with warm fuzzies. Many people, however, feel very differently, and they have good reasons for feeling that way. Most importantly, simply not having to concern yourself with all the quirks of writing browser-based applications, of which there are certainly plenty, is a good one.

In short, if this concept of writing Java code to create a webapp has appeal for you, then check out GWT: `http://code.google.com/webtoolkit`. You most definitely will not be alone in your choice; many people sing the praises of GWT as loudly as they can, and there's certainly no arguing the success of many of the products Google has created with it.

Of course, since you're reading this book, you probably already have an idea which way you intend to go, so we should get to the real topic at hand, and that of course is DWR itself.

Hmm, Are We Forgetting Something? What Could It Be? Oh Yeah, DWR!

Since we're here to talk about DWR, we might as well not delay any further. Consider this a teaser of what is coming in the next chapter.

DWR (`http://getahead.ltd.uk/dwr`), which stands for Direct Web Remoting, is a free, open source product from a small IT consulting group called Getahead, whose members are Joe Walker and Mark Goodwin. DWR is an interesting approach to Ajax in that it allows you to treat Java classes running on the server as if they were local, that is, running within the web browser.

DWR is a Java-centric solution, unlike many of the popular Ajax libraries out there today. Some view this as a limitation, but I for one view it as a plus because it allows for capabilities that probably couldn't be provided in a cross-language way (not without being tremendously more difficult to use and most certainly to maintain for the developers).

If you are familiar with the concept of RPC, or Remote Procedure Call, then you will recognize quickly that DWR is, essentially, a form of RPC (and it had better be, it says so right there in the name: Direct Web **Remoting**!).

MORE ABOUT RPC

RPC is a mechanism by which code executing in one address space, typically one physical computing device, executes a subroutine or procedure on another or a shared network. This is accomplished without the programmer of the calling code explicitly coding for the details of the remote interaction. Stated another way, the programmer writes in essence the same code whether the function being called is on the local system or the remote system. Remote invocation or remote method invocation means the same thing as RPC, but is typically used when discussing object-oriented code.

The basic concepts behind RPC go back to at least 1976, as described in RFC 707. Xerox was one of the first businesses to use RPC, but they called it "Courier." This was in 1981. Sun, shortly thereafter, created what is now called ONC RPC, a form of RPC specifically for UNIX, which formed the basis of Sun's NFS. ONC RPC is still used widely on several platforms today.

Today, there are many analogies to RPC; most well known are probably Java's RMI and Microsoft .NET's Remoting, and there is also a web services implementation of RPC, just to name a few. All of these still share the same basic concepts developed back in 1976, and DWR follows the basic model as well.

In the DWR model, the calling "system" is JavaScript executing in the users' browser, and the system being called is the Java classes running on the server. Just like other forms of RPC, the code you write in JavaScript looks very similar to the code you'd write on the server to execute the same code. There are, of course, some differences as you'd expect given the inherent differences between JavaScript and Java, but by and large it's extremely similar.

To give you a brief preview, consider the following simple class definition:

```
package com.company.app;
public class MyClass {
  String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

If you wanted to call this code via DWR from JavaScript running in the browser, assuming you'd already accomplished all the necessary setup, the following code is all it would take:

```
MyClass.sayHello("Frank", sayHelloHandler);
var sayHelloHandler = function(data) {
  alert(data);
}
```

As I mentioned, and as you can see, the call syntax is very much like you'd do from Java, with the addition of the second parameter, which as it turns out is a callback function that will be executed when the method invocation completes.

In the next chapter, we'll be getting into all the nitty-gritty details of how this works, why the callback is needed, how to configure DWR, etc. For now though, this is a good preview of what DWR is all about because, believe it or not, it doesn't get a whole lot more complex than that!

Summary

If this chapter has seemed like an attempt to brainwash you, that is because, in a sense, it was! Ajax can seem to some people like a really bad idea, but those people tend to only see the problems and completely ignore the benefits. Because of my belief that Ajax is more about philosophy and thought process than it is about specific technologies, it is important to sell you on the ideas underlying it. It is not enough to simply show you some code and hope you agree!

In this chapter, we have looked at the evolution of web development over the past decade and a half or so. We have discussed Ajax and what it means, and seen how it is accomplished. We have discovered the term RIA and talked about why all of this really is a significant development for many people.

We have also discussed how the most important thing about Ajax is not the technologies in use but the mindset and approach to application development that are its underpinnings.

We have seen and played with our first Ajax-based application and have discovered that the code involved is ultimately pretty trivial. We have discussed some alternatives to Ajax and have learned that Ajax has been done for years, but with different techniques and technologies.

We have also discussed how Ajax is more user-driven than developer-driven, which is a significant difference from many advances in the past (and also one good reason why Ajax isn't as likely to go away as easily).

We took a quick look at some alternatives to Ajax, and also some of the libraries out there besides DWR for doing Ajax.

Last but not least, we got our first quick glimpse of DWR itself, setting the stage for the next chapters in which we'll dive into it head first.

I will close with just a few take-away points that succinctly summarize this chapter:

- Ajax is more about a mindset, an approach to developing webapps, than it is about any specific technologies or programming techniques.
- Ajax does in fact represent a fundamental shift in how webapps are built, at least for many people.
- Ajax, at its core, is a pretty simple thing. What we build on top of it may get complex, but the basics are not a big deal.
- RIAs may or may not represent the future of application development in general, but almost certainly do in terms of web development.
- There are a number of alternatives to Ajax that you should not dismiss out-of-hand. Consider the problem you are trying to solve and choose the best solution. Ajax will be it in a great many (and a growing number) of cases, but may not always be. As you have bought this book, however, we'll assume from here on out you want to do Ajax!

CHAPTER 2

Getting to Know DWR

In this chapter, we'll go beyond the basics introduced in Chapter 1 and do a true meet-and-greet with our new best friend, DWR. We'll get to know DWR by beginning with a simple question: why should you use it at all for your Ajax needs in the first place? From there we'll move on to really understanding what DWR is, how it works, and what parts comprise it. We'll then make sure we have a development environment set up that will quickly and easily allow us to play with DWR, and be able to play with the applications we'll build in subsequent chapters. We'll look at all the important facets of using DWR including adding it to a basic webapp, configuring it, and interacting with it. By the end of this chapter, we will have built our first DWR-based application, albeit a very simple one, and will have a firm footing with which to move on to the more advanced topics in the following chapter, and ultimately the applications that make up the remainder of this book.

First Things First: Why DWR at All?

So, your slave-driving Bill Lumbergh[1] of a boss gave you the assignment that you knew had to be coming, given all the hype on the Interwebs[2] these days: take that old, mangy mutt of a webapp that the accounting department depends on and make it fresh, hip, and cool. This of course means throwing a healthy dose of Ajax into the mix.

So, off you go, exploring the myriad libraries and toolkits available for doing Ajax. You even read some material on the details of doing it by hand, but hey, you're a typical lazy . . . err, that is, smart! developer, so reinventing the wheel is something you generally try and avoid, and hence a library it shall be.

But which of the seemingly endless supply of options do you choose? I recently read (although naturally I can no longer find the link!) a tabulation that counted something on the order of 236 libraries available for doing Ajax, and that was **just** those that are Java-specific and free! Needless to say, having options is a good thing, but *also* needless to say, sometimes too much of a good thing can kill you!

---

1. Bill Lumbergh is the iconic boss character in the cult classic *Office Space*. This is one of the few times you'll see me tell you, the reader, that there is something better you could be doing with your time than reading my book, but this is one great movie—go watch it now!
2. *Interwebs* is a slang term used in place of the Internet, most usually the World Wide Web specifically. This term (or its close cousin, Internets) can be heard frequently on *The Colbert Report* on Comedy Central. (The term Internets is usually attributed to George W. Bush, a "Bushism" as it's usually called; although I can't vouch for the voracity of that attribution, it certainly isn't hard to believe!)

So, you investigate some of the most famous names out there: Dojo, Prototype, script.aculo.us, MooTools, APT, just to name a few. They all have something to offer that you like, but none of them strike you as perfect. They all (with the exception of APT, the AjaxParts Taglib in Java Web Parts) also strike you as being fairly similar to each other in terms of how you use their Ajax capabilities. In short, it's all JavaScript that you'll have to write by hand, only the amount and complexity differs; but in the end, they're all a bit low-level, a bit "close to the metal," so to speak.

In the interest of full disclosure, APT is part of Java Web Parts (JWP), a project I originated. It supplies small, largely independent components for Java web developers, things like filters, servlets, taglibs, listeners, etc. APT is by far the most popular part of JWP. It is a taglib that allows you to add Ajax functions to a page without writing **any** JavaScript, and without changing the existing page. You just add some tags in appropriate locations, supply an XML configuration file describing the Ajax functions to perform, and off you go. Have a look here if that sounds interesting: `http://javawebparts.sourceforge.net`. Cruise into the Javadoc and look in the AjaxParts package. All the APT documentation you'd need is right there (the projects' Javadoc is something of a combined Javadoc/user manual).

Fortunately for you, JavaScript is something you're pretty comfortable with. Still, there's no need doing any more of it than necessary. What you really want is a way to access certain business functions of your application (which you've naturally broken out into a nice, clean façade that you can call on independent of the webapp components, right?) that makes your JavaScript not a whole lot different from the existing server-side code in the application.

In short, that's exactly what DWR provides.

DWR was, if not the first, then certainly the first to rise to prominence that provides this approach of treating server-side code like JavaScript running in the browser. Subsequently, other libraries have been created (or some existing ones expanded) to do things much like DWR does, but DWR holds the distinction of doing it first (first in the sense of being well known for it) and many would say still to this day does it best.

So, why should you choose DWR over other options? The answer, of course, as hopefully you've already realized on your own, is maybe you shouldn't. Did I just hear a gasp? Seriously, any developer worth his most likely too small paycheck will always explore his options and recommend the solution that fits the current problem best (ignoring the all-too-common "corporate standards" paradigm where you don't have the freedom to do that). By all means, you should investigate the other options first, write some little pilot apps with each, and see which one fits your requirements the best. DWR will almost certainly rise to the top couple of choices, and in a great many cases it'll fit the bill perfectly. But not all cases. Further, I doubt very much anyone involved with developing DWR would say otherwise. DWR is top-notch and is a great choice in many, perhaps even most, situations, but other options may be just as good, and sometimes another may be better, so don't just settle on DWR (or anything else) without some due diligence going into making a good decision. This is only smart, and hopefully I'm not telling you anything you don't already know and agree with!

DWR: RPC on Steroids for the Web

So, let's get to brass tacks here: what is DWR exactly, where did it come from, and why is it so darned cool to so many people, yours truly included?

First, let's talk about what DWR actually is.

DWR, which stands for Direct Web Remoting, is a combination Java/JavaScript open source library that allows you to simply and easily build Ajax applications without getting into all the details of `XMLHttpRequest` coding. It allows you to call server-side code in a way that looks like it is running locally in the browser from the point of view of the client-side JavaScript that makes the calls. This is exactly what the *direct* part of DWR refers to.

Consider the situation if you have a collection of classes with some functions in each that you wish to be callable from JavaScript. Let's ignore the question of how mechanically you're going to make those calls and just think about what the server-side code would have to look like. Each function would have to be exposed as essentially a service endpoint, and this would likely mean a servlet for each (forgetting about any application framework for just a moment).

Sure, you will likely be smart about it and only have a single servlet involved, but considering that the URI you use to address a given function would necessarily have to be unique, even if it only differed in some parameter you pass in, you can quickly see where this might become a hassle in a hurry. The other important consideration is the protocol you use to make the call. How do you pass parameters to the remote function? How do you marshal the return value(s) into a form you can use in your JavaScript? These are the kinds of details you'll have to deal with yourself, and that's just a taste of what would be involved. (What about error handling? What about security? Argh, the list grows in a hurry, doesn't it?)

DWR releases you from these burdens by taking care of all these details for you. DWR also offers a collection of utility JavaScript functions that provide nearly all the most common functions needed to build dynamic, powerful Ajax applications. "But how?" you ask.

Well, if you know what the term Remote Procedural Call, or RPC, is all about, then you essentially know what DWR is all about too.

Figure 2-1 shows a high-level diagram of what RPC in general is all about (and no, those are not diodes near the top there for you electrically inclined out there!). In RPC, some client code makes a call, most usually via some sort of proxy stub functions or objects. These proxy stubs in turn make a request through the underlying RPC library, whatever implementation that may be. As it's usually the case that the code being called is on another machine, some network protocols get involved as the request is sent to the remote system. There, the underlying RPC library, the server-side component now (since the code you're calling is essentially a server as far as the client code is concerned) receives the request and passes it through some other proxy stub objects, and on to the final endpoint, the procedure being remoted (note that the proxy stub objects on the server side are oftentimes not present; this is a generalized architecture diagram). The *service code*, the remote code being executed by the caller, does its thing and returns the results back through the same flow in reverse.

Note the demarcation line showing the interface provided by the service code. Also note the callout up top noting the apparent call flow. As far as the caller is concerned, the interface to the remote code is identical to what it would be if it were running in the same process space; therefore, the apparent flow from the perspective of the client code doesn't include anything below the demarcation because that all happens transparently. The same is true of the code being remoted.

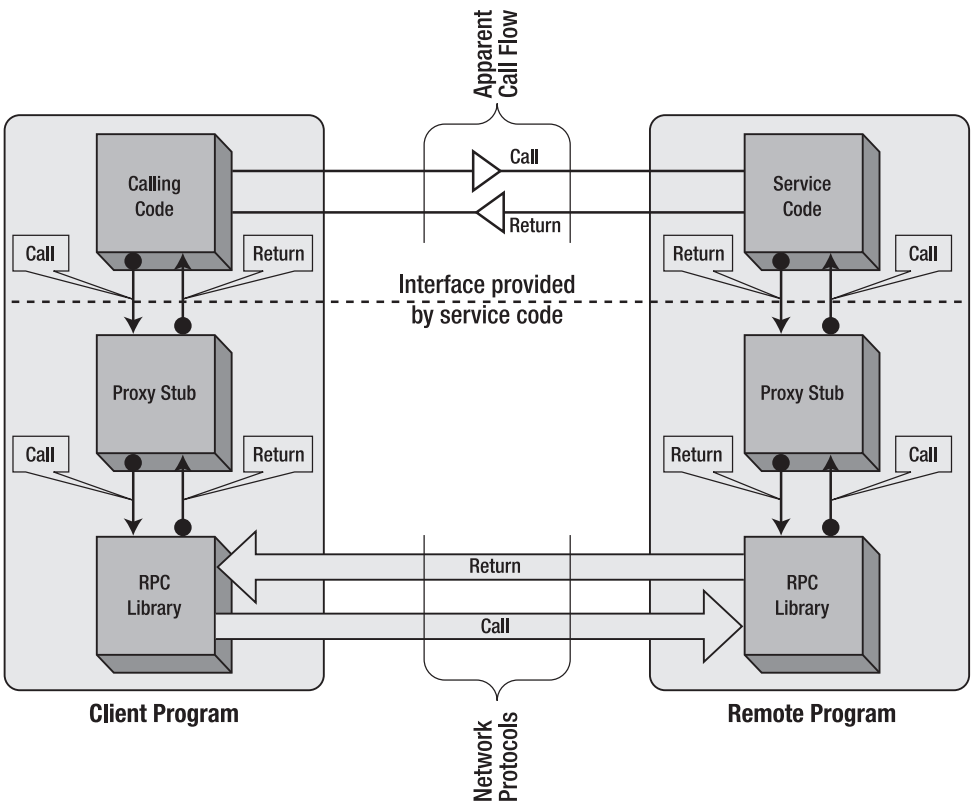


Figure 2-1. *Architectural overview of RPC*

DWR is simply a particular implementation of this diagram. We'll look at the specific parts in the next section, but for now, that's the pertinent point to grep from all this. That and the fact that DWR will save you a tremendous amount of time and energy!

Another key point about DWR is that it's a very nonintrusive tool to use. What I mean by this is that it doesn't force any required architecture down your throat in terms of how you organize your server-side code or your client-side code. Especially in terms of the server-side code, since that's what you'd be remoting with DWR, you are perfectly free to use Spring beans, EJBs, POJOs, servlets, or virtually any other type of Java class you can think of. So long as the DWR code running on the server can get an instance of the object you're calling a function on, then you can use it with DWR.

One other big plus about DWR is that it is very clearly and (nearly) completely documented. You can't say this about every open source project out there, but you'll (almost) never have trouble getting the information you need when it comes to DWR. In addition, the DWR mailing list is an invaluable resource for anyone using the library. The people there, including the developers themselves, are very helpful and very quick to answer any question posted (even the stupid ones from brain-dead authors trying to earn a buck!). This level of support is again something you can't always find in the open source world, and DWR is exemplary in this regard, as well as in documentation.

Speaking of the DWR developers, to complete this section, let's give credit where it's very much due: DWR is the brainchild of one Joe Walker of Getahead (`http://getahead.org`), a small consultancy firm run by Mr. Walker in the United Kingdom. The first release of DWR was in roughly June 2005, and since then it has undergone steady improvement, culminating (so far) with the release of version 2.0 earlier this year. This version brought with it a great deal of new capabilities (which we'll explore as we go along) and made DWR a truly top-notch Ajax library for the Java-facing world (and is there really another world outside that one?).

DWR Architectural Overview

For all the coolness DWR offers, it is architecturally a fairly simple creature. Well, let me be more specific: to the extent you as a developer interact with it, it is pretty simple. Its internal architecture may be somewhat more complex, but by and large we won't be going into that in any great detail in this book. We may touch on parts of it here and there, but you're here to learn how to *use* DWR, not so much to learn the intricate details of how it works its magic. Put another way, we'll be content to leave the wizard behind the curtain alone, but we may take an occasional quick peek here and there!

The DWR documentation has a nice diagram that gives you a high-level picture of what DWR does, and Joe Walker graciously gave permission to include that diagram here, so have a gander at Figure 2-2 if you would.

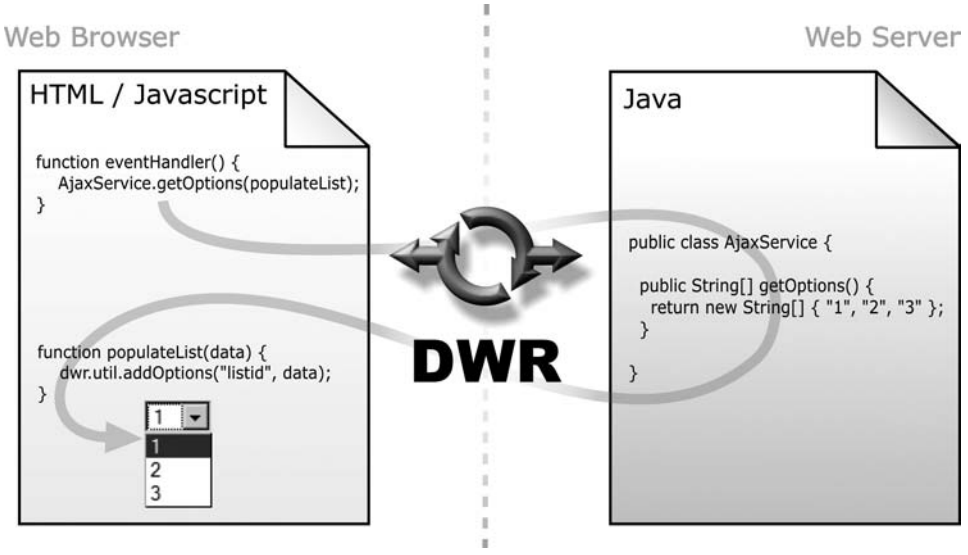


Figure 2-2. *High-level pictoral representation of DWR in action*

This diagram succinctly summarizes how DWR works. You can see on the left-hand side some JavaScript, a function named `eventHandler()`, in which a call to the method `getOptions()` on the object `AjaxService` is made. That code looks like it is executing that method on a local JavaScript object, but it is in fact calling that method on a Java object running on the server. You wouldn't know it by looking at the code though, which is a big part of the coolness of DWR I mentioned.

The parameter passed to the method, `populateList()`, is the name of another JavaScript function, and it is the callback. It will be executed when the call to the remote object returns; hence the reason the circle with the arrows is indicating flow in both directions.

If we try to conceptually map the parts from this diagram into the parts in Figure 2-1, since DWR is a form (or implementation if you will) of RPC, where do things fit? First off, it's pretty obvious that the box marked *Calling Code* on the client side in Figure 2-1 is where the JavaScript in the `eventHandler()` function would be. It's also clear that the box marked *Service Code* on the server side is where the `AjaxService` class would live. But what about the boxes labeled *Proxy Stub* and *RPC Library* on both sides? Those need a little more explanation before we can fit them in.

The main part of DWR, what you can quite correctly think of as its core, is the `DWRServlet` class. This is, by and large, an ordinary Java servlet, but it performs quite a few Herculean feats for us, one of which is automatic generation of JavaScript objects based on Java classes. As you'll see shortly when I present the first real example of using DWR, one of the first things you need to do is include in your web page a link to a JavaScript file. But, here's the twist: it's not actually a JavaScript file! Based on what we see in Figure 2-2, for example, we would include something like this on our page:

```
<script type="text/javascript" src="myApp/dwr/interface/AjaxService.js"> </script>
```

Notice the name of the JavaScript file we're importing here? It's the name of the Java class in the diagram! What happens when the page that has this line on it is interpreted in the browser is that a request is made for the `AjaxService.js` resource at the location `myApp/dwr/interface`. This URI maps to the `DWRServlet`. The servlet then dynamically inspects the `AjaxService` Java class and generates JavaScript representing it and returns that JavaScript. So now, in global scope on the page, we have a JavaScript object named `AjaxService`. Guess what? That's our client-side proxy stub right there! So now we know what maps into the *Proxy Stub* box on the client side of Figure 2-1.

So, what about the *RPC Library* box in Figure 2-1 on the client side of the diagram? That's actually easy: it's the JavaScript portion of DWR itself. As you'll see shortly, there's another `<script>` that is required on your page (which also maps to the servlet) that is the client-side portion of DWR that transparently does all the heavy Ajax lifting.

To complete the picture, let's figure out what fits in the *Proxy Stub* and *RPC Library* boxes on the server side of Figure 2-1. The *RPC Library* box mapping should now be pretty obvious: it's the `DWRServlet` itself (and any helper classes it may use). In the case of DWR, there really isn't anything in the *Proxy Stub* box on the server side, however (remember I mentioned that some RPC implementations wouldn't have that when we discussed it earlier), so we can effectively remove that box from the diagram in our minds' eye.

So, at this point we can conceptually understand what the pieces of DWR are at a high level, and how they fit into the diagram of RPC. But, it's always better to **see** the big picture rather than just conceptualize it, so Figure 2-3 does just that. In it you can see the overarching flow of a DWR call, with notations describing which parts of the flow map to what parts of the RPC diagram in Figure 2-1.

Our next step will be to set up the development environment so we can play with DWR, and then finally, at long last, write our first application using DWR.

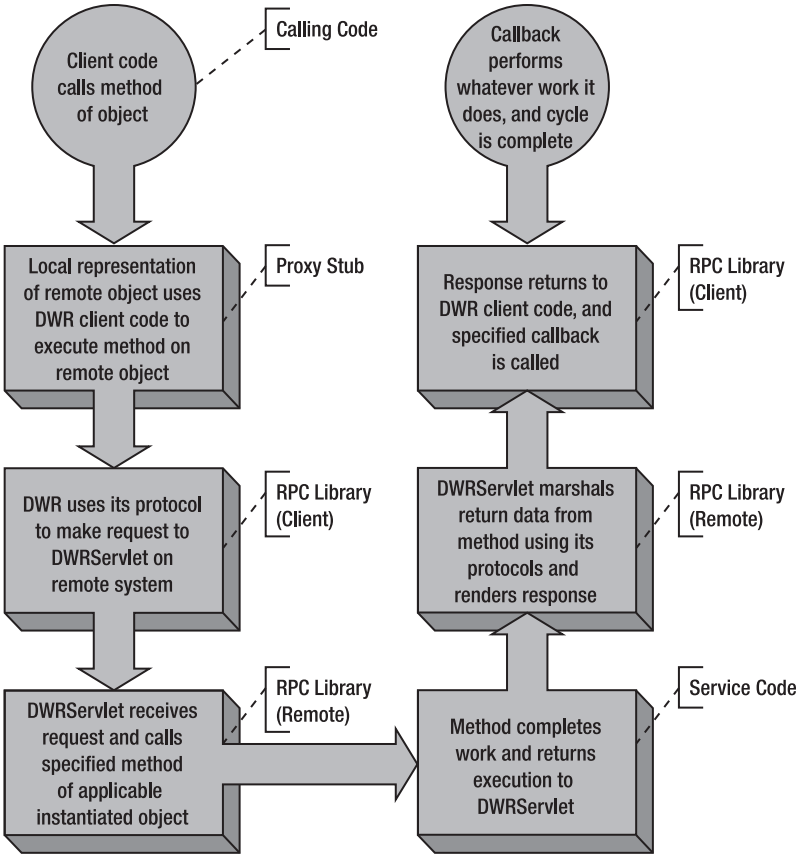

Client code calls method of object
Calling Code
Local representation of remote object uses DWR client code to execute method on remote object
Proxy Stub
DWR uses its protocol to make request to DWRServlet on remote system
RPC Library (Client)
DWRServlet receives request and calls specified method of applicable instantiated object
RPC Library (Remote)
Method completes work and returns execution to DWRServlet
Service Code
DWRServlet marshals return data from method using its protocols and renders response
RPC Library (Remote)
Response returns to DWR client code, and specified callback is called
RPC Library (Client)
Callback performs whatever work it does, and cycle is complete

Figure 2-3. *The basic DWR request flow, mapped to the RPC architecture parts*

Getting Ready for the Fun: Your DWR Development Environment

To play with the applications and sample code throughout this book, you will need to have a properly configured development environment. DWR 2.0 requires JDK 1.3 or higher, and a servlet engine that supports servlet spec 2.2 or higher. DWR works in numerous containers and application servers including Tomcat, WebLogic, WebSphere, JBoss, Jetty, Resin, and GlassFish. One caveat is that if you have a web server in front of your app server, and the web server alters the URLs, DWR may (and probably will) fail.

Although that's what DWR requires at a minimum, one of the goals of any good author (I'm tryin' folks!) is to have the example code work as expected with no problems. To help achieve that, I very much suggest matching your environment to that in which the code was developed. The following is the environment configuration I used. You may well already be good to go based on the minimums DWR requires, and you can skip this section if your environment meets the requirements, but if your environment doesn't match what's described

next to a reasonable approximation, then I very much suggest taking the time to set things up to match this list, just to avoid any problems.

- *Sun Java SDK 1.6.0_03 (otherwise known as 6u3, or 6 update 3, with a build number of 1.6.0_03-b06)*: If you do not have this already installed, please visit `http://java.sun.com/javase/downloads/index.jsp` and download it. It sure looks to me like Sun is continuing with their master plan of making the version numbering so confusing as to be rendered utterly meaningless to virtually everyone on planet Earth, so just download the latest Java 6 (which is 1.6) that is there. At the time I wrote this, it was SDK 6u3. The install is a simple matter of running the installer and following the prompts. I personally like to install my SDK into `c:\java`, no version number, so that I can swap in another SDK at any time without updating anything that may be pointing to it. That's completely up to you, however; you can install it however you wish.

Verify that you have the SDK properly installed by going to a command prompt and typing **java -version**. You should see something like this:

```
C:\>java -version
java version "1.6.0_03"
Java(TM) SE Runtime Environment (build 1.6.0_03-b06)
Java HotSpot(TM) Client VM (build 1.6.0_03-b06, mixed mode)
```

This also assumes you have your classpath pointing to the SDK directory's `bin` subdirectory, which is covered in the last bullet point in this list.

- *Apache Tomcat 6.0.13*: This can be downloaded from `http://tomcat.apache.org/download-60.cgi`. I suggest getting the ZIP version of the Core distribution, which can simply be unzipped, no need to install anything. To start the server, simply navigate to the `<TOMCAT>/bin` directory and execute `startup.bat`. You can then access the root webapp by going to `http://localhost:8080`, which should result in the page shown in Figure 2-4. Note that you will likely see a warning at startup indicating the Apache Tomcat Native Library was not found. This is perfectly safe, but if you'd like to get rid of it, you can download the native library and make it available to Tomcat.
- *Apache Ant 1.7.0*: Grab it at `http://ant.apache.org/bindownload.cgi`. This will only be required if you want to build the projects from source, which you don't have to, but then again that kind of is the point of the book!

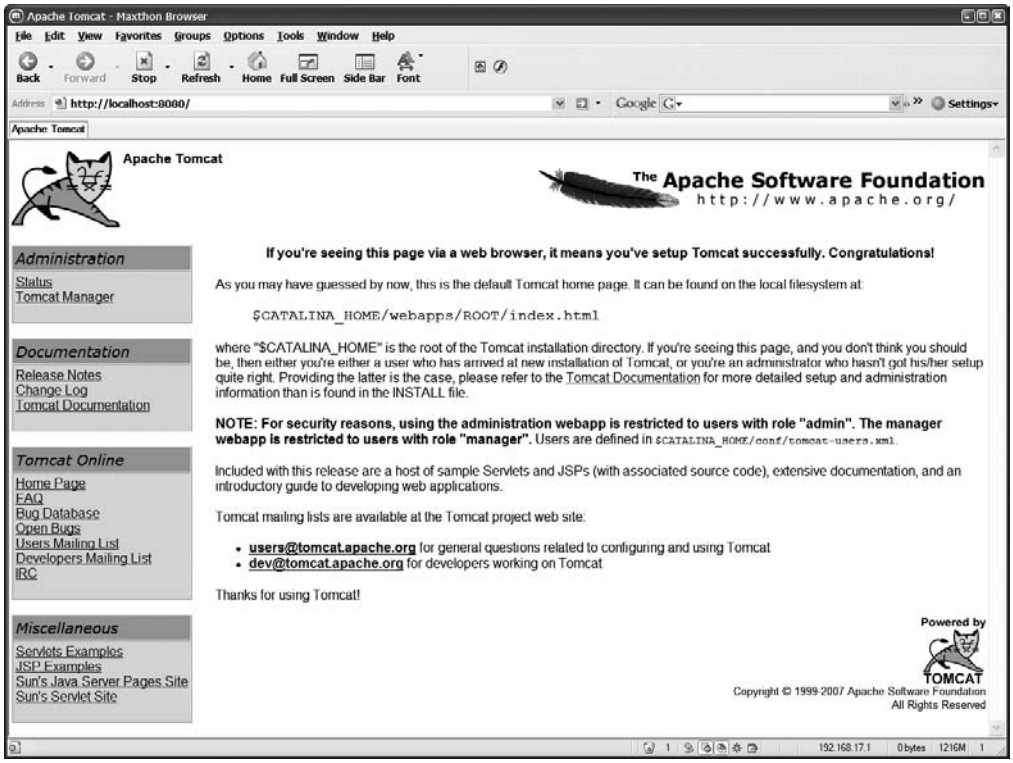

Apache Tomcat - Maxthon Browser
File Edit View Favorites Groups Options Tools Window Help
Back Forward Stop Refresh Home Full Screen Side Bar Font
Address http://localhost:8080/
Google
Settings
Apache Tomcat
Apache Tomcat
The Apache Software Foundation
http://www.apache.org/
Administration
Status
Tomcat Manager
Documentation
Release Notes
Change Log
Tomcat Documentation
Tomcat Online
Home Page
FAQ
Bug Database
Open Bugs
Users Mailing List
Developers Mailing List
IRC
Miscellaneous
Servlets Examples
JSP Examples
Sun's Java Server Pages Site
Sun's Servlet Site
If you're seeing this page via a web browser, it means you've setup Tomcat successfully. Congratulations!
As you may have guessed by now, this is the default Tomcat home page. It can be found on the local filesystem at:
$CATALINA_HOME/webapps/ROOT/index.html
where "$CATALINA_HOME" is the root of the Tomcat installation directory. If you're seeing this page, and you don't think you should be, then either you're either a user who has arrived at new installation of Tomcat, or you're an administrator who hasn't got his/her setup quite right. Providing the latter is the case, please refer to the Tomcat Documentation for more detailed setup and administration information than is found in the INSTALL file.
NOTE: For security reasons, using the administration webapp is restricted to users with role "admin". The manager webapp is restricted to users with role "manager". Users are defined in $CATALINA_HOME/conf/tomcat-users.xml.
Included with this release are a host of sample Servlets and JSPs (with associated source code), extensive documentation, and an introductory guide to developing web applications.
Tomcat mailing lists are available at the Tomcat project web site:
users@tomcat.apache.org for general questions related to configuring and using Tomcat
dev@tomcat.apache.org for developers working on Tomcat
Thanks for using Tomcat!
Powered by
TOMCAT
Copyright © 1999-2007 Apache Software Foundation
All Rights Reserved
192.168.17.1
0 bytes
1216M

Figure 2-4. *The Tomcat startup page*

Verify that you have Ant properly installed by going to a command prompt and typing **ant -version**. You should see something like the following:

```
C:\>ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
```

This also assumes you have your classpath pointing to the Ant directory's `bin` subdirectory, which is covered in the last bullet point in this list.

- *Two environment variables set*: `JAVA_HOME`, which should point to the directory your SDK was installed to, and `ANT_HOME`, which should point to the directory Ant was installed to. In addition, you'll want to add `<JAVA_HOME>/bin` and `<ANT_HOME>/bin` to your path. Lastly, as a recommendation, I highly suggest having **no** `CLASSPATH` environment variable set. This will greatly reduce the chance of any kind of classpath issues biting you.

While it may be the case that newer versions of any of these will work fine, and in fact even older versions may to some extent, I highly suggest matching your environment to these versions if they aren't already. This will prevent the small possibility that any of the applications don't work for some reason.

In terms of IDE, you can use whatever you like. My own personal preference is for nothing but a good text editor (I recommend UltraEdit for Windows, `www.ultraedit.com`) and a command line. Choice of an IDE is a highly personal matter; some people such as me prefer no IDE at all, others swear by Eclipse or NetBeans or IDEA or some other product. I'm not here to try and sway your opinion in any direction, but for the sake of consistency, everything in this book will assume no IDE at all and will assume everything is done from a command prompt. I will also be assuming a Windows environment, although there's no reason you can't do everything in a *nix environment.

USE THE SOURCE, LUKE!

At this point I'd like to point out that all of the source code for this book, including ready-to-run WAR files for all the projects, can be downloaded from the Apress web site at `www.apress.com`. Once there, click the Source Code/Download link and find this book in the list, which will bring you to the download page.

Please note that this is not just a suggestion, it is all but required to move forward! From this point out, I will assume that you have downloaded the source code, have unzipped it, and therefore have it available to refer to. This will not only save you oodles of time and pain from typing far more than you should have to, but it will also all but assure that the code you're trying to run is in good working order.

A Simple Webapp to Get Us Started

Our first order of business at this point is to throw together a quick, simple little webapp that we'll add some DWR to in the next section. The complete app, including DWR, is included in the source download package, but if you'd like to build it from scratch (and this is one of the few times I'd suggest doing that instead of using the prebuilt code), follow along here, it's nothing complicated.

Getting the Lay of the Land: Directory Structure

First, go into the `<TOMCAT>/webapps` directory and create a new directory named `simpleapp`. In that directory, create a directory named `WEB-INF`. This is, of course, the typical Java webapp directory structure, which should be old hat to you for sure, but we'll go through it just the same to be certain.

Inside the `WEB-INF` directory, create a directory named `lib` and another named `src`. The directory `lib` is where our JAR files that the app depends on to run are located, and the `src` directory is where the server-side source code for the app will go. Lastly, inside the `src` directory, create another directory named `app`, which is where our Java source files will be.

By the end of this, you should have the directory structure shown in Figure 2-5 underneath `<TOMCAT>/webapps`.

Figure 2-5. *Directory structure of our first webapp (eventually) using DWR*

The next thing we need to do is get the JAR files this application will need and add them. First, when we want to compile the source code for the application, we'll need access to the Servlet APIs. These are present in a JAR file named, not surprisingly, `servlet-api.jar`. This can be found in the `<TOMCAT>/lib` directory. Copy that JAR file over into `simpleapp/WEB-INF/src` (we don't want to put it in `simpleapp/WEB-INF/lib` because it's provided to our webapp via Tomcat, so it would be redundant if added to the classpath for the application and could actually cause conflicts).

From Code to Executable: Ant Build Script

The next step is to create an Ant build script file that we can use to compile our source code (which we haven't written yet, of course). The build script can be seen in Listing 2-1. You should save this to a file named `build.xml` and save it to `simpleapp/WEB-INF/src`.

Listing 2-1. *The Ant Build Script for the Application*

```
<project name="simpleapp" basedir="." default="build">


  <!-- Classpath for build. -->
  <path id="classpath">
    <pathelement path="servlet-api.jar" />
  </path>


  <!-- The default target.  Compile source, build WAR. -->
  <target name="build">
    <delete dir="../classes/com" />
    <mkdir dir="../classes/com" />
    <javac srcdir="." destdir="../classes" debug="true"
     debuglevel="lines,vars,source">
      <compilerarg value="-Xlint:unchecked" />
      <classpath refid="classpath" />
    </javac>
    <delete file="simpleapp.war" />
    <jar destfile="simpleapp.war" basedir="../../" />
  </target>


</project>
```

This is a very simple build script that performs two simple tasks. First, it compiles the source code found in `simpleapp/WEB-INF/src/app` and puts the resultant Java `.class` files in `simpleapp/WEB-INF/classes`. Second, it creates a WAR file for deployment to a servlet container. If this is your first exposure to Ant, I suggest taking a few moments to read the documentation available on the Ant web site since that's a bit out of the scope of this book. The scripts for the projects to come won't be a whole lot more complicated than this one, but they will in fact be somewhat more so, and while you can get away without ever looking at them, it's better to understand it all for the sake of completeness.

The `<delete>` and `<mkdir>` tasks you'll notice don't delete `WEB-INF/classes`, they delete `WEB-INF/classes/com`. The reason is that in many of the applications in this book, there are files in `WEB-INF/classes` that can't be deleted if the application is to function properly.

Application Configuration: web.xml

Now that we've essentially put together the infrastructure for the application, let's get to the application components themselves. First up is the `web.xml` file, as shown in Listing 2-2. Save this to `simpleapp/WEB-INF`.

Listing 2-2. *The* `web.xml` *File for the Application*

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<web-app id="simpleapp" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">


  <!-- Our servlet, the target of the form submission. -->
  <servlet>
    <servlet-name>MathServlet</servlet-name>
    <servlet-class>app.MathServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MathServlet</servlet-name>
    <url-pattern>/MathServlet</url-pattern>
  </servlet-mapping>


  <!-- Default page to load in context. -->
  <welcome-file-list>
```

```
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>


</web-app>
```

As you can see, there's not much going on here. In fact, it's only two things: a default document declaration so that when we access `http://localhost:8080/simpleapp`, which will be the URL to use to try this out, we'll get `index.jsp` right away, and a servlet declaration, which will be the target of the form submission that we'll see soon. In case you haven't noticed by now, by creating this directory structure directly in `<TOMCAT>/webapps`, we're effectively deploying the application automatically. We don't need to do anything else to make this application available via Tomcat. This is a fast, efficient way to work on a webapp that can really save you time during the development process. However, you'll also notice that the build script creates a WAR file. So, if you prefer not working directly in the Tomcat directory structure, you can simply do all this elsewhere, and copy the WAR file into `<TOMCAT>/webapps`. On startup, Tomcat will see this WAR file and unpack and deploy your application automatically (in fact, Tomcat by default is configured to unpack and deploy WAR files on the fly, so even if Tomcat is running, assuming you haven't disabled automatic deployment, a WAR file dropped in this directory will be deployed automatically). I'm all about choice, so however you're more comfortable, feel free!

The Markup: index.jsp

Next up is our single page, `index.jsp`, as shown in Listing 2-3. Please save this to the `simpleapp` directory.

Listing 2-3. *The* `index.jsp` *Page*

```
<html>
  <head>
    <title>simpleapp</title>
  </head>
  <body>
    <%
      Integer answer = (Integer)request.getAttribute("answer");
      // If there's an answer attribute in request, we just performed an
      // operation, so display the result.
      if (answer != null) {
        Integer a = (Integer)request.getAttribute("a");
        String op = (String)request.getAttribute("op");
        Integer b = (Integer)request.getAttribute("b");
    %>
          <span><h1>Result: <%=a%> <%=op%> <%=b%> = <%=answer%></h1></span>
    <%
      }
    %>
    Please enter two numbers, select an operation, and click the equals button:
```

```
    <br><br>
    <form action="MathServlet" method="post">
      <input type="text" name="a" size="4">
       
      <select name="op">
        <option value="add">+</option>
        <option value="subtract">-</option>
        <option value="multiply">*</option>
        <option value="divide">/</option>
      </select>
       
      <input type="text" name="b" size="4">
       
      <input type="submit" value="=">
    </form>
  </body>
</html>
```

As long as this isn't your first exposure to JSPs, this should be readily understandable to you. Notice that I'm using a scriplet block instead of JSTL or something more elegant because this is frankly very simple code, and I felt it was cleaner to not bring in extra dependencies and have more to talk about. To explain the code you see quickly: when the form is submitted, it will hit the `MathServlet`, which we'll see next. That will call the `MathDelegate` class, which is what does the actual math. The servlet then puts a number of attributes into the `request` object, including the two numbers (the attributes `a` and `b`) the user submitted, as well as the mathematical symbol for the operation the user requested (the `op` attribute), and of course the answer. So, if the `answer` attribute is found, then we grab the two numbers and the operation from the request object. Then, we just output the result, which is the complete expression, answer and all. Certainly not rocket science!

On the Server Side: MathServlet.java

Only one step remains, and that's to create the Java classes for the `MathServlet` and the `MathDelegate` that the preceding JSP makes use of. Please save Listing 2-4 and Listing 2-5 (the latter shown in the next section) to the `simpleapp/WEB-INF/src/app` directory.

Listing 2-4. *The* `MathServlet` *Class*

```
package app;


import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;


/**
 * This servlet is the target of the form submission.  It dispatches to the
```

```
 * MathDelegate class to do the actual work, then forwards to index.jsp.
 */
public class MathServlet extends HttpServlet {


  /**
   * The typical doPost() method of a servlet.
   *
   * @param  request          The request object.
   * @param  response         The response object.
   * @throws ServletException If anything goes wrong.
   * @throws IOException      If anything goes wrong.
   */
  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    // Get the numbers to operate on, and the operation to perform, as
    // specified by the caller.
    int a = Integer.parseInt(request.getParameter("a"));
    int b = Integer.parseInt(request.getParameter("b"));
    String op = request.getParameter("op");
    MathDelegate mathDelegate = new MathDelegate();
    int answer = 0;

    // Call the MathDelegate to perform the appropriate operation, and store
    // a math symbol representing the operation in request.
    if (op.equalsIgnoreCase("add")) {
      answer = mathDelegate.add(a, b);
      request.setAttribute("op", "+");
    } else if (op.equalsIgnoreCase("subtract")) {
      answer = mathDelegate.subtract(a, b);
      request.setAttribute("op", "-");
    } else if (op.equalsIgnoreCase("multiply")) {
      answer = mathDelegate.multiply(a, b);
      request.setAttribute("op", "*");
    } else if (op.equalsIgnoreCase("divide")) {
      answer = mathDelegate.divide(a, b);
      request.setAttribute("op", "/");
    }

    // Add the two numbers and the answer to request, so our JSP can display it.
    request.setAttribute("a", new Integer(a));
    request.setAttribute("b", new Integer(b));
    request.setAttribute("answer", new Integer(answer));

    // Forward to index.jsp.
    RequestDispatcher dispatcher =
```

```
      getServletContext().getRequestDispatcher("/index.jsp");
    dispatcher.forward(request, response);


  } /// End doPost().


} // End class.
```

This is a pretty ordinary servlet really. You'll quickly note that this isn't exactly production-quality code! All of the checks you'd perform to avoid errors are left out, among other things, but that is just for the sake of keeping the listing relatively short in print. In any case, you should get the general gist of it in short order. It grabs the two numbers the user submitted, along with the operation the user selected, and calls on the `MathDelegate` class to perform the desired operation. It then puts the numbers, the operation, and the answer in the `request` object as attributes, and finally forwards back to `index.jsp`, which as we've seen already will display the result.

The Workhorse: MathDelegate.java

The `MathDelegate` class, which `MathServlet` calls on, is our next stop, and it's very simplistic, even more so than the servlet. For those of you who prefer a more visual approach to learning, Figure 2-6 shows the UML class diagram for the `MathDelegate` class.

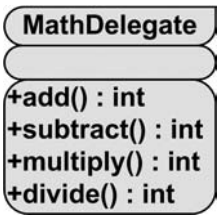


Figure 2-6. *UML diagram of the* `MathDelegate` *class*

In Listing 2-5, you can see the code for the class in its entirety.

Listing 2-5. *The* `MathDelegate` *Class*

```
package app;


/**
 * A class to perform basic mathematical operations on two numbers.
 */
public class MathDelegate {
```

```
/**
 * Add two numbers.
 *
 * @param  a The first number.
 * @param  b The second number.
 * @return   The result of the operation.
 */
public int add(int a, int b) {

  return a + b;

} // End add().


/**
 * Subtract two numbers.
 *
 * @param  a The first number.
 * @param  b The second number.
 * @return   The result of the operation.
 */
public int subtract(int a, int b) {

  return a - b;

} // End subtract().


/**
 * Multiply two numbers.
 *
 * @param  a The first number.
 * @param  b The second number.
 * @return   The result of the operation.
 */
public int multiply(int a, int b) {

  return a * b;

} // End multiply().


/**
 * Divide two numbers.
 *
 * @param  a The first number.
 * @param  b The second number.
```

```
   * @return   The result of the operation.
   */
  public int divide(int a, int b) {

    if (b != 0) {
      return a + b;
    } else {
      return 0;
    }

  } // End divide().


} // End class.
```

Four simple methods, one for each of our four basic math operations, and that's it. Again, not exactly robust code: the only check is the obvious divide-by-zero possibility; otherwise we let everything go through. And again, it's all about making a simple, clean example app (and besides, mathematics and I haven't frequently been on speaking terms, and this is about as advanced as I'm willing to get!).

It's Alive: Seeing It in Action

At this point we have ourselves a complete webapp. Well, nearly so: we still need to build it. To do so, go to a command prompt in the `simpleapp/WEB-INF/src` directory, enter **ant**, and press Return. Ant should execute the build script in that directory, compiling the `MathServlet` and `MathDelegate` classes and creating a WAR file.

Once you are ready, just start Tomcat as previously described in the section "Getting Ready for the Fun: Your DWR Development Environment" and navigate to the URL `http://localhost:8080/simpleapp`, and you should see the page shown in Figure 2-7.

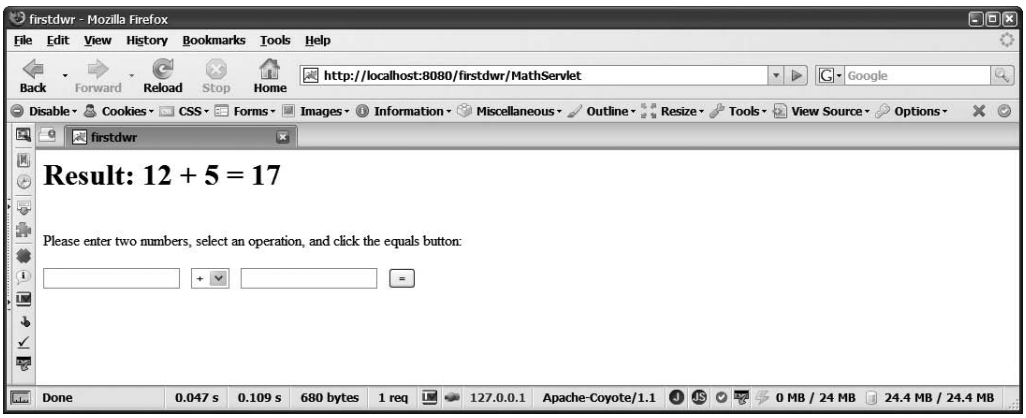


Figure 2-7. *Our simple webapp in action, simple as it is!*

At this point, you can enter two numbers (or not, and get an exception if you prefer . . . this isn't robust production-quality code by any stretch!), select an operation, and click the equals button. Note that the page refreshes with each click. It is being rendered each and every time. Clearly, this is a perfect place for Ajax, and what's that I hear? Is that the pitter-patter of horse hooves? Yes, indeed it is, it's DWR to the rescue, 299 Spartans in tow![3]

Adding DWR to the Mix

So, we have a simple little webapp, and we have one problem with it: that annoying page refresh. You also may not so much like the fact that there's a little more server-side code than you might like, i.e., that servlet in the mix. Let's add DWR to this thing and see how the formula changes.

The first step you should take is to copy the entire `simpleapp` directory and rename the copy to `firstdwr`. You'll have a new webapp to hack without messing up what you just created.

The next thing you'll need to do is add the DWR JAR itself. This too is included in the source code download archive, or you can download it from the DWR web site at `http://getahead.org/dwr`. Copy the `dwr.jar` file into `firstdwr/WEB-INF/lib` (this one is needed at runtime as well as compile time, so it has to be on the application's classpath, and hence in the `lib` directory).

DWR.JAR, DWR.WAR, AND THE SOURCE

If you go to the address specified here, you may notice that there are three things you can download. One is the `dwr.jar` itself. The other is the complete source code for DWR. The third is named `dwr.war`, and if you have a few moments, you may want to spend them grabbing that too. Drop it into Tomcat's `webapps` directory and fire it up. This webapp is a little demo app demonstrating a number of neat DWR functionalities. It actually contains a number of example applications as well, including a chat application, a quick address entry form where address details are populated automatically based on postal code, and a real-time stock quote demo using something called General Interface (GI), which is an open source UI library with DWR at its core.

3. In 480 B.C., the Battle of Thermopylae took place. This military engagement pitted King Leonidas of Sparta, a city in Greece, with 300 of his finest soldiers (and 700 Thespian volunteers, which many people forget . . . Thespian here does not refer to actors, but to the Greek city Thespiae) in a last-stand battle against the invading Persian Empire, lead by King Xerxes, at the pass of Thermopylae, a very narrow valley pass. Leonidas and his troops held off the Persian army, which reportedly numbered upwards of 80,000 eventually (in three waves of attack actually: 10,000 the first day, 20,000 the second day, and 50,000 the final day). Xerxes' troops were finally victorious, but not without heavy losses. This battle is often cited as the best example of how well-trained, disciplined troops using natural resources to their fullest can overcome seemingly unbeatable odds (the fact that they lost notwithstanding). Oh yeah, the recent movie *300* was based on this battle. Pretty good flick if you don't mind lots of violence and men with abs you and I will never have!

Now for the first round of changes required. First, we need to remove the `MathServlet`, since we won't need it (DWR will essentially take its place), and we therefore also need to add an entry for the `DWRServlet` into `web.xml`. Listing 2-6 shows the modified `web.xml` code.

Listing 2-6. *The Modified* `web.xml` *File, All DWR-ified*

```
<?xml version="1.0" encoding="ISO-8859-1"?>


<web-app id="firstdwr" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">


  <!-- The DWR servlet. -->
  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
   </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
   </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>


  <!-- Default page to load in context. -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>


</web-app>
```

Don't worry too much about the details of the `DWRServlet` setup; we'll get into that in the next section. For now, just note that the servlet is mapped to all paths beginning with `/dwr`. This will be important when we look at the updated JSP.

Speaking of the JSP, we have to modify that as well. Listing 2-7 shows those updates.

Listing 2-7. *The Updated* `index.jsp` *File*

```
<html>
  <head>
    <title>firstdwr</title>
    <script type="text/javascript" src="dwr/interface/MathDelegate.js"></script>
    <script type="text/javascript" src="dwr/engine.js"></script>
    <script>
      var a = 0;
      var b = 0;
      var op = "";
      function doMath() {
        a = document.getElementById("numA").value;
        b = document.getElementById("numB").value;
        op = document.getElementById("op").value;
        if (op == "add") {
          MathDelegate.add(a, b, doMathCallback);
          op = "+";
        } else if (op == "subtract") {
          MathDelegate.subtract(a, b, doMathCallback);
          op = "-";
        } else if (op == "multiply") {
          MathDelegate.multiply(a, b, doMathCallback);
          op = "*";
        } else if (op == "divide") {
          MathDelegate.divide(a, b, doMathCallback);
          op = "/";
        }
      }
      var doMathCallback = function(answer) {
        document.getElementById("resultDiv").innerHTML = "<h1>" +
          "Result: " + a + " " + op + " " + b + " = " + answer + "</h1>";
      }
    </script>
  </head>
  <body>
    <span id="resultDiv"></span>
    Please enter two numbers, select an operation, and click the equals button:
    <br><br>
    <input type="text" id="numA" size="4">
     
    <select id="op">
      <option value="add">+</option>
      <option value="subtract">-</option>
      <option value="multiply">*</option>
      <option value="divide">/</option>
    </select>
     
```

```
    <input type="text" id="numB" size="4">
     
    <input type="button" value="=" onClick="doMath();">
  </body>
</html>
```

A couple of changes have been made, beginning with the removal of the JSP scriplet block and the HTML form (the elements of the form are still there, but notice they are no longer contained in a `<form>` element). The form isn't really necessary at this point because we're going to be grabbing the field values manually and using them as parameters to a method call. Also note that the fields no longer have `name` attributes, they now have `id` attributes instead. This is so that the JavaScript code that has been added can access them directly by `id`.

Next, note that the submit button has been changed to a plain old button element. When the button is clicked, the JavaScript `doMath()` function will be called, which is a new addition, as is the entire `<script>` block up top, and the two `<script>` imports before it.

Speaking of the imports, there are two of them. One of them, the one that pulls in the file `engine.js`, is the actual DWR client-side code. The other, `dwr/interface/MathDelegate.js`, is the client-side proxy stub for the `MathDelegate` class. Notice that both of these URIs begin with `dwr`. Remember that all URIs beginning with `dwr` will be handled by the `DWRServlet`. Hence, the two pieces of JavaScript that are being imported are being generated by `DWRServlet`, one of them (the proxy stub) dynamically.

Now, let's move on to the JavaScript that has been added after those two imports. This in effect takes the place of the JSP scriplet from the original version. First, we have three variables in global scope, two for the numbers the user enters (`a` and `b`) and one for the operation the user performs (`op`). When the equals button is clicked, `doMath()` is called, and the first thing it does is to retrieve those three pieces of information. It then branches based on the operation. In each branch, a call to the appropriate method of `MathDelegate` is made, much like the `MathServlet` did in the original version. The `MathDelegate` object that the method is being called on is the proxy stub object that DWR generated to represent the server-side `MathDelegate`. The code in the four branches also sets the `op` variable to the applicable math symbol, so we can display the equation properly as in the original version.

Did you notice anything suspicious about the four calls to the methods of `MathDelegate`? What's up with that third parameter? It doesn't appear in the method signature of the actual `MathDelegate` class. As it turns out, this parameter specifies a JavaScript function that will be "called back" when the call to the real server-side `MathDelegate` object returns. This is because JavaScript by nature is an asynchronous language, while Java is synchronous. This means there is no mechanism to make the remote call via Ajax and wait for it to return (other than specifying the call is to be synchronous, which you can do at a low level with the `XMLHttpRequest` object; however, it blocks **everything** in the browser, meaning it becomes unresponsive while awaiting the request's completion, so it's generally a bad idea and not quite what we want). Therefore, the callback mechanism is the logical answer.

Once the request completes, the callback function, `doMathCallback()`, is called. Notice the parameter to the method. This is a representation of the return type of the method called, which DWR has marshaled for us. This can be virtually any data type, intrinsic or custom, and in this case it's just a simple integer. Once we have the value, the code simply constructs the appropriate markup to display the equation, and inserts it into a new `<span>` element named `resultDiv`.

There is one new piece to the puzzle, and that's the `dwr.xml` file that we have to create to tell DWR a few things. Create the `dwr.xml` file from the code in Listing 2-8 and save it to `firstdwr/WEB-INF`.

Listing 2-8. *The New* `dwr.xml` *File*

```
<?xml version="1.0" encoding="UTF-8"?>


<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">


<dwr>
  <allow>
    <create creator="new" javascript="MathDelegate">
      <param name="class" value="app.MathDelegate" />
    </create>
  </allow>
</dwr>
```

The `dwr.xml` file's only job here (although it can do more, as we'll see later) is to tell DWR what classes it is allowed to remote, in this case just the `MathDelegate` class (convenient, since that's the only one in the app!). You can specify the name of the JavaScript object representing the server-side object, although most of the time it's likely to be the same, as it is here. The other attributes and options we'll go over in the next section.

So, what do we wind up with after all these changes? Well, I'm not going to show a screenshot here because it would look identical to Figure 2-6. However, running the webapp, you should notice a difference: the page does not refresh when you click the equals button. Instead, the result appears where it appears in the original version, but without changing anything else on the page. It's also a bit faster because of this, as you can probably perceive (it's not a huge difference if you're running these both locally, but you should be able to notice by the absence of page flickering when you click the button due to the refresh not being done). And with that, you've now written your first DWR application, even if an incredibly simplistic one!

The DWR Test/Debug Page

Before we get into the details of DWR configuration, I want to point out another facility that DWR gives you automatically when you use it in your application, and that is the test/debug page. If you access the URL `http://localhost:8080/firstdwr/dwr/index.html`, you'll see the page shown in Figure 2-8.

Now granted, that doesn't look like any big deal. But, go ahead and click the link and you'll see something infinitely more interesting! Since DWR knows about our `MathDelegate` class, it can automatically generate a page that shows us what methods are available on the object, how to access them, and what imports we'll need, and even provides a method to actually test these methods! Figures 2-9 and 2-10 show what this page looks like (the page scrolls larger

than I could capture at once, that's why it's broken into two figures, but understand both of these represent a single page).

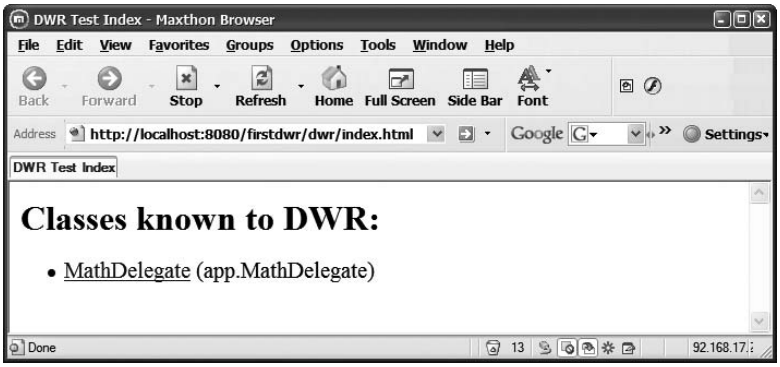

DWR Test Index - Maxthon Browser
File Edit View Favorites Groups Options Tools Window Help
Back Forward Stop Refresh Home Full Screen Side Bar Font
Address http://localhost:8080/firstdwr/dwr/index.html
Google Settings
DWR Test Index
Classes known to DWR:
MathDelegate (app.MathDelegate)
Done

Figure 2-8. *The DWR test/debug page*

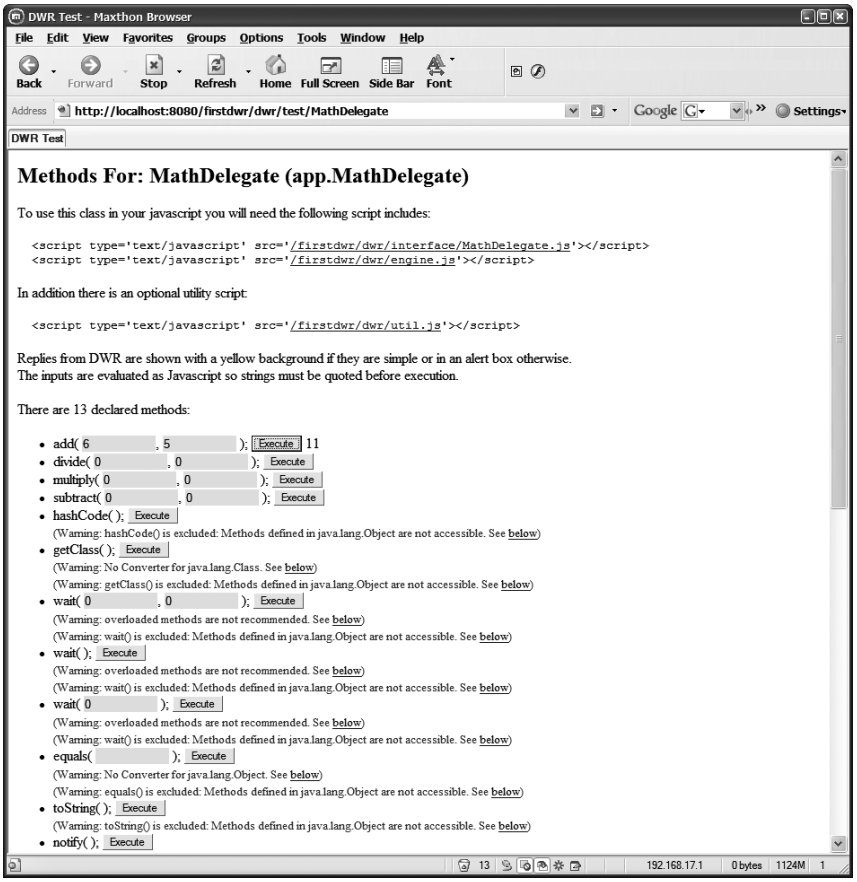

DWR Test - Maxthon Browser
File Edit View Favorites Groups Options Tools Window Help
Back Forward Stop Refresh Home Full Screen Side Bar Font
Address http://localhost:8080/firstdwr/dwr/test/MathDelegate
Google Settings
DWR Test
Methods For: MathDelegate (app.MathDelegate)
To use this class in your javascript you will need the following script includes:
<script type='text/javascript' src='/firstdwr/dwr/interface/MathDelegate.js'></script>
<script type='text/javascript' src='/firstdwr/dwr/engine.js'></script>
In addition there is an optional utility script:
<script type='text/javascript' src='/firstdwr/dwr/util.js'></script>
Replies from DWR are shown with a yellow background if they are simple or in an alert box otherwise.
The inputs are evaluated as Javascript so strings must be quoted before execution.
There are 13 declared methods:
add(6 , 5); Execute 11
divide(0 , 0); Execute
multiply(0 , 0); Execute
subtract(0 , 0); Execute
hashCode( ); Execute
(Warning: hashCode() is excluded: Methods defined in java.lang.Object are not accessible. See below)
getClass( ); Execute
(Warning: No Converter for java.lang.Class. See below)
(Warning: getClass() is excluded: Methods defined in java.lang.Object are not accessible. See below)
wait(0 , 0); Execute
(Warning: overloaded methods are not recommended. See below)
(Warning: wait() is excluded: Methods defined in java.lang.Object are not accessible. See below)
wait( ); Execute
(Warning: overloaded methods are not recommended. See below)
(Warning: wait() is excluded: Methods defined in java.lang.Object are not accessible. See below)
wait(0); Execute
(Warning: overloaded methods are not recommended. See below)
(Warning: wait() is excluded: Methods defined in java.lang.Object are not accessible. See below)
equals( ); Execute
(Warning: No Converter for java.lang.Object. See below)
(Warning: equals() is excluded: Methods defined in java.lang.Object are not accessible. See below)
toString( ); Execute
(Warning: toString() is excluded: Methods defined in java.lang.Object are not accessible. See below)
notify( ); Execute
192.168.17.1 0 bytes 1124M

Figure 2-9. *The* `MathDelegate` *test/debug page, part 1*

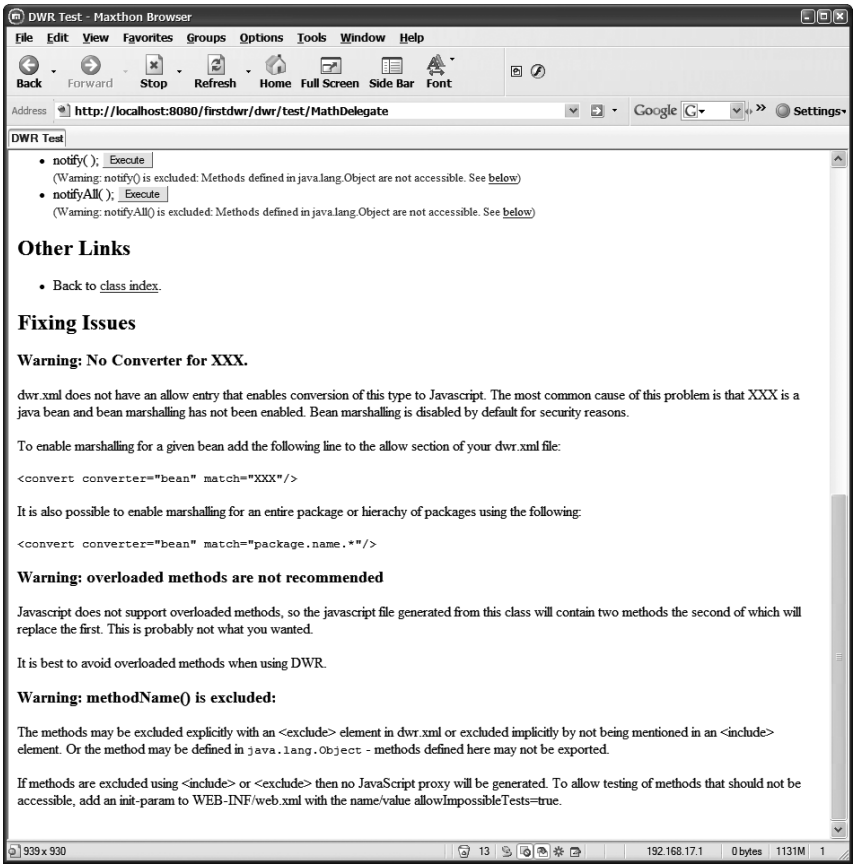

DWR Test - Maxthon Browser
File Edit View Favorites Groups Options Tools Window Help
Back Forward Stop Refresh Home Full Screen Side Bar Font
Address http://localhost:8080/firstdwr/dwr/test/MathDelegate
Google Settings
DWR Test
notify( ); Execute
(Warning: notify() is excluded: Methods defined in java.lang.Object are not accessible. See below)
notifyAll( ); Execute
(Warning: notifyAll() is excluded: Methods defined in java.lang.Object are not accessible. See below)
Other Links
Back to class index.
Fixing Issues
Warning: No Converter for XXX.
dwr.xml does not have an allow entry that enables conversion of this type to Javascript. The most common cause of this problem is that XXX is a java bean and bean marshalling has not been enabled. Bean marshalling is disabled by default for security reasons.
To enable marshalling for a given bean add the following line to the allow section of your dwr.xml file:
<convert converter="bean" match="XXX"/>
It is also possible to enable marshalling for an entire package or hierachy of packages using the following:
<convert converter="bean" match="package.name.*"/>
Warning: overloaded methods are not recommended
Javascript does not support overloaded methods, so the javascript file generated from this class will contain two methods the second of which will replace the first. This is probably not what you wanted.
It is best to avoid overloaded methods when using DWR.
Warning: methodName() is excluded:
The methods may be excluded explicitly with an <exclude> element in dwr.xml or excluded implicitly by not being mentioned in an <include> element. Or the method may be defined in java.lang.Object - methods defined here may not be exported.
If methods are excluded using <include> or <exclude> then no JavaScript proxy will be generated. To allow testing of methods that should not be accessible, add an init-param to WEB-INF/web.xml with the name/value allowImpossibleTests=true.
939 x 930
192.168.17.1
0 bytes
1131M

Figure 2-10. *The* `MathDelegate` *test/debug page, part 2*

You can even see in Figure 2-9 where I've actually tested the `add()` method. The result is shown next to the Execute button. Any of the methods can be tested in this fashion, without you having to have written a single bit of code!

This page also gives you some helpful notes at the bottom about various situations that you will want to be aware of. For instance, the note about overloaded methods is a good one to keep in mind. As it says, because JavaScript doesn't support overloaded methods (you can fake it, but it's not true overloading), this is something you need to be aware of when writing your server-side classes.

All in all, this is a fantastic facility that DWR offers that you will want to keep in mind for sure. It can save you all sorts of time and effort when getting things working.

Configuring DWR Part 1: web.xml

We'll begin looking at how to configure DWR, and the logical starting point is the `web.xml` file. The absolute minimum you could add to an existing `web.xml` to get DWR up and running is this:

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

While you have the freedom to map the servlet to any path you'd like, or use extension-based mapping even, I very much suggest using this standard mapping since it's logical and will be one less difference you have to mentally parse when you're looking for help on the Web.

Beyond this basic setup, there are a number of init parameters you can specify. They are summarized in Table 2-1.

Table 2-1. *The* `DWRServlet` *Init Parameters*

Parameter	Description
`allowGetForSafariButMakeForgeryEasier`	In Safari 1.*x*, there is a browser bug that causes the body contents of `POST` requests to be dropped. This is, to put it mildly, a bit of a problem in most cases. Setting this to `true` will make DWR work in that browser regardless. This defaults to `false`.
`crossDomainSessionSecurity`	Setting this to `false` will enable requests from other domains other than the one that app is hosted on. This can be a rather significant security risk, so you should change this with caution and be sure you understand all the consequences. The default value is `true`.
`allowScriptTagRemoting`	Script tag remoting is a method of Ajax where a `<script>` tag is dynamically added to the page. The browser then goes and retrieves the specified JavaScript file. This file will be in the form of a function call with data (usually JSON or some similar data structure) as the argument to the function. The function already exists on the page, so what you in effect have set up is a mechanism where you can request a resource, and that resource when inserted into the page causes a JavaScript callback function to be executed, passing it the data you were interested in retrieving. This is a useful approach to Ajax because it is one of the few ways you can do cross-domain Ajax calls (the `XMLHttpRequest` object, for instance, has what's known as a *same domain* policy, meaning it will only allow requests to the domain the host document was served from). Setting this to `true` (which is the default) allows DWR to use this method if necessary.
`debug`	Setting this value to `true` enables the test/debug page previously discussed. By default, this is `false`.

Parameter	Description
scriptSessionTimeout	This sets the amount of time in milliseconds before a script session times out. The default is 1800000, or 30 minutes.
maxCallCount	This sets the maximum number of requests allowed in a batch. The default is 20. The purpose of this setting is to help reduce the possibility of overloading the server by mistake, and to help avoid Denial of Service attacks.
activeReverseAjaxEnabled	When set to true, the polling and Comet reverse Ajax techniques are enabled. This defaults to false.
maxWaitingThreads	This sets how many threads the servlet will keep in an active state for handling requests. The default is 100.
maxPollHitsPerSecond	When using the polling reverse Ajax method, this is the maximum number of requests per second allowed. The default value is 40.
preStreamWaitTime	This is the maximum amount of time in milliseconds to wait before opening a stream to reply. By default, this is 29000.
postStreamWaitTime	This is the maximum amount of time in milliseconds to wait after opening a stream to reply. By default, this is 1000.
[Interface Name]	This is used to override parts of DWR without having to build it from source. The default implementation is the default.
ignoreLastModified	By default this is false. DWR uses Last-Modified headers to help the client determine when a request for a given resource should be made, thereby reducing those requests due to smart caching. Setting this to true will disable this capability.
scriptCompressed	DWR has the ability to perform a very simple compression of the returned JavaScript, and setting this parameter to true enables that compression (by default it's false). Along with this is an undocumented (although official) parameter named compressionLevel. There are three levels of compression: none, normal, and ultra. See the Javadoc for the JavascriptUtil class for more details.
sessionCookieName	To support session, DWR can use URL rewriting to include the session cookie with each request it makes. By default, the standard JSESSIONID cookie name is used, but this parameter allows you to use a different cookie name if you wish (of course, your container will have to know to use the alternate name as well).

Continued

Table 2-1. *Continued*

Parameter	Description
`normalizeIncludesQueryString`	When using reverse Ajax, pages with differing query strings are considered the same page. However, for some sites, this assumption is not correct. Setting this parameter to `true` (the default being `false`) will force DWR's reverse Ajax implementation to consider the query string when comparing URLs.
`overridePath`	Some servlet containers, when a web server is in front of them, will alter the path for the request, which will cause DWR to set the wrong locations for its Ajax calls. Using this parameter, you can override the path DWR uses to account for this. By default, this is not used.

As you can see, there are quite a few configuration options available on the `DWRServlet` itself. In most cases you'll find the defaults to be just what you need, and that will generally be the case in the applications throughout this book. However, as exceptions come up, I'll explain the reasoning behind the parameters used. The one exception is the `debug` parameter, which you'll see used virtually every time. This is just too handy a tool to turn off anywhere but a production environment!

You should also be aware that there is an alternate controller servlet that you can choose if you wish to use the annotations capabilities DWR offers. Annotations are covered in the next chapter, and I'll give you the full details about the other servlet when we get there.

You probably noticed a couple of references to something called *reverse Ajax*, but what exactly is that? We're going to get into it in detail in Chapter 3, but as a bit of a preview, reverse Ajax is a technique that allows you to asynchronously send data from the **server** to the **client**. In point of fact, it's not **truly** doing that, but it very much gives the appearance that it is, and really for all practical purposes you can treat it as if it were really doing it. Like I said, don't worry about it much now, we'll get to this in the next chapter.

Configuring DWR Part 2: dwr.xml

The next topic of conversation is the DWR-specific configuration housed in the `dwr.xml` file. DWR actually offers two ways of configuring it, through Java 5 (and up) annotations and the `dwr.xml` file. You can in fact do everything you do with `dwr.xml` with annotations instead, or you can supplement `dwr.xml` with annotations, but one step at a time here—let's look at the configuration file approach first.

The basic structure of the `dwr.xml` file can be shown concisely in image form, and Figure 2-11 is just such an image.

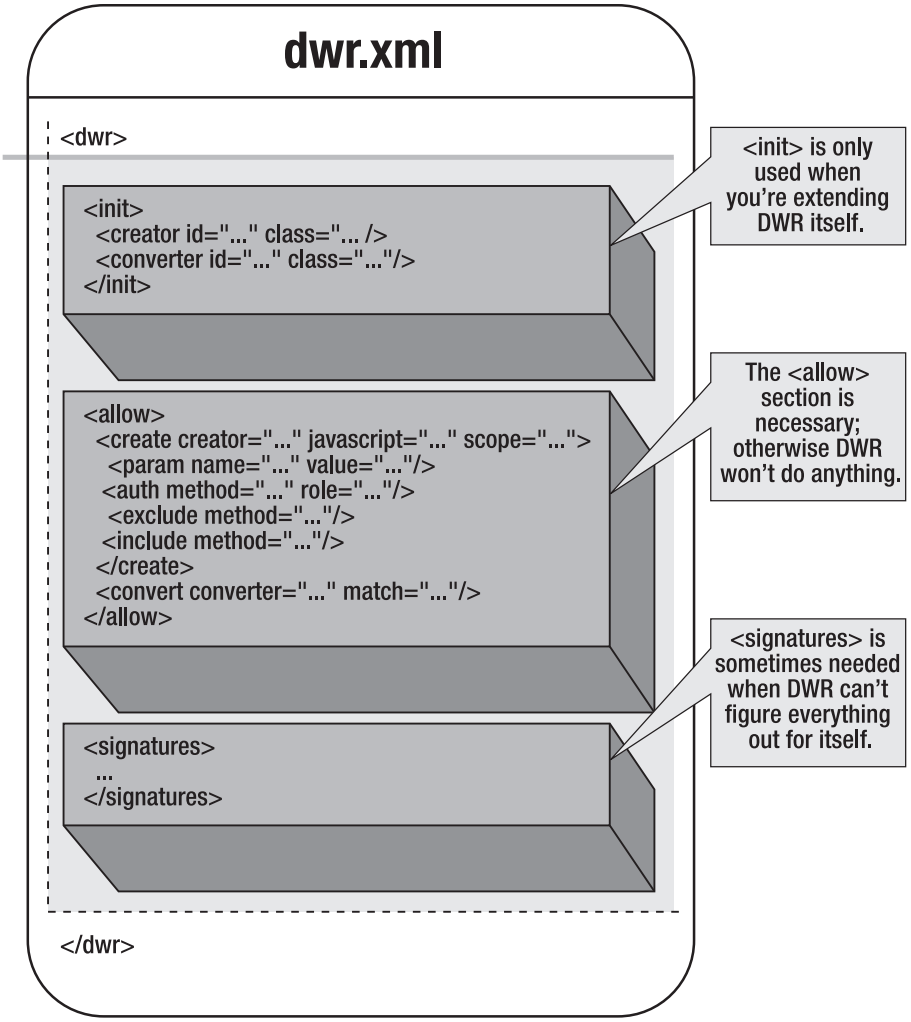


Figure 2-11. *Graphical representation of the structure of* `dwr.xml`

If you would instead prefer all the gory details, then Listing 2-9 is just the ticket for you: it's the `dwr.xml` DTD. Of course, this is the kind of thing you should only read if you're having trouble sleeping (I kid . . . reading a DTD, especially when commented well as this one is, gives you a one-stop shop kind of reference for a given XML file).

Listing 2-9. *The Cure for Insomnia: The* `dwr.xml` *DTD File*

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
Top level conviguration element.
-->
```

```
<!ELEMENT dwr (
    (init?), (allow?), (signatures?)
)>

<!--
A list of all the classes to configure as part of dwr at startup time.
-->
<!ELEMENT init (
    (creator | converter)*
)>

<!--
Define a new method of creating objects for use by Javascript.
We don't just allow access to any object and some may need special code to
get a reference to them.
-->
<!ELEMENT creator EMPTY>
<!--
@attr id The unique name by which create elements refer to us.
@attr class The fully qualified name of a class that implements Creator.
-->
<!ATTLIST creator
    id ID #REQUIRED
    class CDATA #REQUIRED
>

<!--
Define a new way of converting between javascript objects and java objects.
Many classes can have default conversion mechanisms but some require more
custom conversion
-->
<!ELEMENT converter EMPTY>
<!--
@attr id The unique name by which create elements refer to us.
@attr class The fully qualified name of a class that implements Converter.
-->
<!ATTLIST converter
    id ID #REQUIRED
    class CDATA #REQUIRED
>

<!--
Security: we must define which classes we are allowed to access because a
free-for-all will be very dangerous.
-->
<!ELEMENT allow (
    (create | convert)*
)>
```

```
<!--
Allow the creation of a class, and give it a name in javascript land.
A reference to a creator is required as are some parameters specific to each
creator that define the objects it allows creation of.
It would be nice to make the creator and IDREF rather than a CDATA, since it
refers to an element defined elsewhere, however we allow multiple dwr.xml
files and we might refer to one in another file.
-->
<!ELEMENT create (
    (param | include | exclude | auth)*
)>
<!--
@attr creator The id of the creator to use
@attr javascript The name of the object to export to the browser
@attr scope The scope of the created variable. The default is page.
-->
<!ATTLIST create
    creator CDATA #REQUIRED
    javascript CDATA #REQUIRED
    scope (application|session|request|page) #IMPLIED
>

<!--
Some elements (currently only create although there is no hard reason why
convert elements should not be the same) need customization in ways that we
can't predict now, and this seems like the only way to do it.
-->
<!ELEMENT param (#PCDATA)>
<!--
@attr name The name of the parameter to this creator
@attr value The value to set to the names parameter
-->
<!ATTLIST param
    name CDATA #REQUIRED
    value CDATA #IMPLIED
>

<!--
A creator can allow and disallow access to the methods of the class that it
contains. A Creator should specify EITHER a list of include elements (which
implies that the default policy is denial) OR a list of exclude elements
(which implies that the default policy is to allow access)
-->
<!ELEMENT include EMPTY>
<!--
@attr method The method to include
-->
```

```
<!ATTLIST include
    method CDATA #IMPLIED
>

<!--
See the include element
-->
<!ELEMENT exclude EMPTY>
<!--
@attr method The method to exclude
-->
<!ATTLIST exclude
    method CDATA #IMPLIED
>

<!--
The auth element allows you to specify that the user of a given method must be
authenticated using J2EE security and authorized under a certain role.
-->
<!ELEMENT auth EMPTY>
<!--
@attr method The method to add role requirements to
@attr role The role required to execute the given method
-->
<!ATTLIST auth
    method CDATA #REQUIRED
    role CDATA #REQUIRED
>

<!--
Allow conversion of a class between Java and Javascript.
A convert element uses a previously defined converter and gives a class match
pattern (which can end with *) to define the classes it allows conversion of
It would be nice to make the converter and IDREF rather than a CDATA, since it
refers to an element defined elsewhere, however we allow multiple dwr.xml
files and we might refer to one in another file.
-->
<!ELEMENT convert (
    (param)*
)>
<!--
@attr converter The id of the converter to use
@attr match A class name to match for conversion
-->
<!ATTLIST convert
    converter CDATA #REQUIRED
    match CDATA #REQUIRED
>
```

```
<!--
If we are marshalling to collections, we need to be able to specify extra
type information to converters that are unable to tell from reflection what to
do. This element contains some Java mthod definitions
-->
<!ELEMENT signatures (#PCDATA)>
```

OK, before we go too far, let's get some terminology under our belts. First, any javabean that is to be remoted will require a *creator,* which knows all the details of how to create a bean of a given type. The parameters that are passed to a method and returned from methods will require a *converter*, which knows all the details of how to marshal the Java types to and from JavaScript types.

Built-in Creators and Converters

DWR comes with a number of creators to deal with a number of bean types, and these are listed in Table 2-2.

Table 2-2. *The Built-in DWR Creators*

Creator	Description
`new`	Uses the Java `new` operator. This is likely the one you'll use most often as it allows you to remote beans of virtually any type, but it does no setup of the bean prior to making the remote call, as other creators may do.
`none`	Does not create objects. This can be necessary if (a) the object is assumed to already exist, or (b) the call is to a static method.
`spring`	Gives access to beans through the Spring Framework.
`jsf`	Uses objects from JSF, or Java Server Faces.
`struts`	Uses Struts ActionForm beans.
`pageflow`	Gives access to a PageFlow from Beehive or WebLogic.
`ejb3`	An experimental creator to give access to EJB3 session beans. This code should not be seen as production quality until it has undergone more testing, and has an official maintainer.

DWR also comes with a number of converters to handle most of the basic types you'll encounter day to day, and better still, you don't need to declare them anywhere—they are always enabled and ready for you to use. These converters include the following:

- `boolean`
- `byte`
- `short`
- `int`
- `long`
- `float`

- double
- char
- java.lang.Boolean
- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.lang.Character
- java.math.BigInteger
- java.math.BigDecimal
- java.lang.String
- A converter for converting arrays
- Two converters for dealing with collections, namely the collection and map converters
- The enum converter, for dealing with Java enumerations
- The bean and object converters, which are similar in that they will convert a javabean to and from a JavaScript associative array, but differ in that the object converter works with object members directly, rather than through accessor and mutator methods.
- Date converters that marshal between JavaScript dates and various Java date types (java.util.Date, java.sql.Date, java.sql.Times, and java.sql.Timestamp).

The <init> Section

The <init> section comes into play when you need to have your own custom creators and converters. This is something we'll see in detail later in the chapters to come, but basically it amounts to creating a class that implements DWR's Converter interface for custom converters, or the Creator interface for creators. You then declare this class in the <init> section, giving it an ID that you can reference in the rest of the configuration.

The <allow> Section

Now we get to the <allow> section, which is without question where you'll be spending most of your time in the dwr.xml file. This section is where you tell DWR what beans can be remoted, and also how your own custom beans that are parameters or return types will be marshaled to and from JavaScript.

It all starts with adding a <create> element to the <allow> section. You'll need to specify which creator to use by setting the creator attribute, and probably most of the time you'll use the value new. You'll also need to set the javascript attribute. This is the name of the object you'll interact with in your client code. You may optionally set the scope attribute, which is very much like the scope you deal with in servlet programming, meaning the values are things like application, request, session, and page. The default value is page, and most of the time this is what you'll want anyway. Note that session requires the use of cookies or URL rewriting.

The <param> elements, which are children of the <create> element, are additional pieces of information the creator needs to do its job. What parameters are available are specific to each creator. For instance, the "new" creator needs to know what class to instantiate, hence you add a <param name="class" value="...">. I'll introduce the parameters as we use the creators, but you can check them out now in the DWR documentation on the DWR web site if you wish.

The <auth> element, also a child of the <create> element, is for integrating with J2EE container-managed security. This will be looked at in the next chapter.

Finally, the <include> and <exclude> elements, again children of the <create> element, define a list of methods the creator will allow or deny access to. These elements are mutually exclusive, so if you want your default access policy for a given class to be to allow all methods to be called, then you would specify a list of methods with the <exclude> element. Conversely, if you want your default policy to disallow access to all methods, then set method names with the <include> element. This allows you more fine-grained control over what methods of a given class can be remoted.

All of the code that is discussed in the next section can be seen in action in the webapp seconddwr, which is part of the source download for this book. One interesting thing to see there that isn't discussed here explicitly is how the call to SigTestClass.convertNames() is done, given that its parameter is a List. Note the JavaScript array notation used to represent that list.

The <convert> element, which is a child of the <allow> element, defines that DWR is allowed to convert a particular class to and from JavaScript, using a specified converter. For instance, let's say you have a class, call it ClassA, and it has a method, creatively called methodA:

```
package app;


/**
 * A class used to demonstrate the bean converter.
 */
public class ClassA {


  /**
   * This method returns an instance of MyBean, which will be converted to a
   * JavaScript object by the bean converter.
```

```
   *
   * @return An instance of MyBean.
   */
  public MyBean methodA() {

    MyBean myBean = new MyBean();
    myBean.setMyField("A value has been set in myField");
    return myBean;

  } // End methodA().


} // End class.
```

This method returns a type `MyBean`, which is a custom class like so:

```
package app;


/**
 * A test bean used to demonstrate the bean converter.
 */
public class MyBean {


  /**
   * A test field.
   */
  private String myField;


  /**
   * Mutator for myField.
   *
   * @param inMyField New value for myField.
   */
  public void setMyField(final String inMyField) {

    myField = inMyField;

  } // End setMyField().


  /**
   * Accessor for myField.
   *
   * @return Value of myField.
```

```
   */
  public String getMyField() {

    return myField;

  } // End getMyField().


} // End class.
```

You want to be able to return an instance of this class from `methodA`, so DWR has to know how to convert it to JavaScript. There are a number of built-in converters, one of which is the *bean* converter. This is actually disabled by default because there are some security concerns. Simply stated, DWR takes the tact that none of your code should be touched in any way without you explicitly giving permission to do so. Therefore, clients are not allowed to remote any classes that you do not specify it being OK to remote, nor will they convert any object that you do not specify it being OK to convert. So, you will have to enable this converter, if you wish to use it, in the `<allow>` section like so:

```
  <allow>

    <convert converter="bean" match="app.MyBean" />

    <create creator="new" javascript="ClassA">
      <param name="class" value="app.ClassA" />
    </create>
```

Now, DWR will be able to marshal the `MyBean` instance into a suitable JavaScript object, and vice versa. The `match` attribute specifies the class the converter will function on. You can also specify a package here, which will cover all classes in the package. You also could specify a match value of "`*`", which would allow conversion of all javabeans throughout your application. This is clearly something you want to be careful doing though because it opens up the possibility that you expose a method that can return a particular type of object you'd rather not allow a client access to. Imagine, for instance, some method that probably should be private, but mistakenly isn't, that returns some internal data structure that contains sensitive information. Internally, that may not be too big of a problem, but allowing a remote client to get at that structure probably isn't something you want to do, so using that wildcard capability can be dangerous if you aren't super careful!

The <signatures> Section

The final section of the `dwr.xml` file to examine is the `<signatures>` portion. This is an optional section that nine times out of ten you won't need at all. When DWR is preparing to call the specified method of the remote object, it does some reflection magic to determine the types of parameters the method expects, and likewise what its return type is. It does this to determine what converters will be involved. However, it is sometimes the case that the information is not available through reflection. In these cases, you have to give DWR a hand and specify this information for it, and that's where the `<signatures>` element comes into play.

Let's say you have a class `SigTestClass`, like so:

```
package app;


import java.util.ArrayList;
import java.util.List;


/**
 * A class to demonstrate the signature section of dwr.xml.
 */
public class SigTestClass {


  /**
   * Method to return a List of names after they have been converted to all
   * uppercase.
   *
   * @param  inList A List of names.
   * @return        The values from inList, converted to uppercase.
   */
  public List<String> convertNames(final List<String> inList) {

    List<String> outList = new ArrayList<String>();
    for (String name : inList) {
      outList.add(name.toUpperCase());
    }
    return outList;

  } // End convertNames().


} // End class.
```

In this case, DWR doesn't know the underlying types of the objects in `inList` (remember that DWR doesn't require Java 5, so the fact that generics are used here to convey that information is irrelevant and not useful to DWR), and that's information it'll need to properly convert to JavaScript. So, to solve this problem, we add the `<signatures>` section to our config file as follows:

```
<signatures>
  <![CDATA[
  import java.util.List;
  import app.SigTestClass;
  SigTestClass.convertNames(final List<String> inList);
  ]]>
</signatures>
```

This looks like Java 5 code because it includes generics, doesn't it? In fact, DWR includes its own syntax parser, so Java 5 is **not** required. As you can see, you need to include imports for the types involved and the signature of the method with the appropriate generic-type notation to indicate the appropriate types. If you have more than one method you need to specify a signature for, you simply add on to the list:

```
<signatures>
  <![CDATA[
  import java.util.List;
  import app.SigTestClass;
  SigTestClass.convertNames(final List<String> inList);
  SomeOtherClass.someOtherMethod(Map<String> ms);
  ]]>
</signatures>
```

As I stated earlier, this section is usually not needed, but it's great to know the capability is there for the times you inevitably do need it and would otherwise be pulling your hair out!

One interesting note: did you notice that there is no return type specified in the method signature? This is not an oversight. Due to a bug in the parser in DWR 1.0, specifying the return type will often result in an error, so it is best to leave this out. At the time of this writing, I could not verify if that bug has been corrected in 2.0 or not, so it is probably best to continue to leave the return type out if you need to specify a signature.

Interacting with DWR on the Client

Now that we've seen how to configure DWR, let's move on to what's more fun for us coder geeks, and that's actually writing some code to use DWR. We'll start by looking at the code you write on the client, since that's where you're likely to spend 99 percent of your time. (You can in fact interact with DWR on the server, as we'll see in the next section, but it frankly doesn't come up very often, especially if you've written your remote object code cleanly and generically . . . remember, those remote objects shouldn't, a vast majority of the time, know they are being called by DWR at all.)

Basic Call Syntax

We've already seen basically how things work, but let's review. Here's the call to the `MathDelegate` class from the firstdwr application:

```
MathDelegate.add(a, b, doMathCallback);
var doMathCallback = function(answer) {
  document.getElementById("resultDiv").innerHTML = "<h1>" +
    "Result: " + a + " " + op + " " + b + " = " + answer + "</h1>";
}
```

As you will recall, the first line is a call to a method of a client-side proxy object. This object then, via the DWR client code, makes an Ajax request that winds up in the `DWRServlet`. The servlet then instantiates an instance of the real `MathDelegate` class, and calls the `add()` method on it, passing in the parameters (which have been converted as necessary). The method returns, the servlet performs any required conversion, and then returns the response

to the client. The DWR code gets the Ajax request and calls the `doMathCallback()` function, which does its thing. We now know, of course, that DWR magically creates this proxy object dynamically, and we also know how the creator instantiates the `MathDelegate` object, and we know how the converters factor in to marshal the incoming parameters from JavaScript to Java and from Java to JavaScript again with regard to the method's return value.

In this approach, we are passing the callback function to the proxy stub. You could inline the function as well, like so:

```
MathDelegate.add(a, b, function(answer) {
  document.getElementById("resultDiv").innerHTML = "<h1>" +
    "Result: " + a + " " + op + " " + b + " = " + answer + "</h1>";
});
```

Both of these are functionally equivalent and really just come down to personal preference. Mine is that the first approach, where the callback is a stand-alone function, is easier to read, but this can vary by situation and is again a matter of preference.

Call Metadata Object Approach

There is, however, a whole other approach you can take to writing this code, and that's to use a metadata object. Here's what it looks like:

```
MathDelegate.add(5, 5, { callback : doMathCallback });
function doMathCallback(inResp) {
  alert(inResp);
}
```

Here, we are passing an object to the proxy stub that is an associative array with some known elements that the client-side DWR code can use. Here, we've specified the callback handler function using the `callback` attribute. Once again, you could choose to inline that callback function in place of `doMathCallback` like so:

```
MathDelegate.add(
  7, 7, { callback : function(inResp) { alert(inResp); } }
);
```

Is there any reason to choose one approach over the other? Yes, in fact there sometimes is. When you use the metadata object approach, you have the opportunity to pass extra call parameters that you otherwise couldn't. For instance, you can pass a `timeout` and an `errorHandler` element in the metadata object. The `timeout` element specifies the amount of time to wait before the request times out, and the `errorHandler` element allows you to specify a function that will handle known errors (e.g., things like 404 errors).

If all you need to specify is a callback, then you may as well go with the first approach since it is, to most people I suspect, a cleaner syntax. If you have to pass other information, or want to make it easier to do so in the future, then using the metadata object approach may be better. The choice really is yours though.

A Word on Some Funky Syntax

As a consequence of DWR's call syntax being so flexible, you can sometimes get into some funky syntax where it's difficult to determine what parameters are what. For instance, take this line of code:

```
SomeClass.someMethod({ timeout : 3 }, { callback : someFunction });
```

In this line, the first parameter is a parameter to be passed to the remote method, and the second is a metadata object. The problem is, determining which is which isn't necessarily straightforward, and is even less so given the naming chosen. DWR uses a known sequence of checks to make the determination as follows:

- If the first parameter to the proxy stub, or the last parameter to the proxy stub, is a function, then the function is the callback, and there is assumed to be no metadata object; all other parameters are arguments to be passed to the remote object.
- Else, if the last parameter is a JavaScript object that contains a callback member that is a function, then this object is taken to be a metadata object, and all other parameters are arguments to the remote object's method.
- Else, if the first parameter is null, it is assumed that there is simply no callback function, and all other parameters are arguments to the remote method. This is useful if there is no return value from the remote method. One caveat to this, however, is that if the last parameter is also null, this will cause a warning.
- Finally, if the last parameter passed to the proxy stub method is null, then there is effectively no callback function.
- If any of these conditions are not true, then the call is malformed, and an error will be thrown.

This information is really valuable insofar as when an error (or a warning) occurs, you will be equipped to understand what the cause is and remedy it.

Setting Beans on a Remote Object

Let's say you have a bean as follows:

```
package app;


import java.util.ArrayList;
import java.util.List;


/**
 * A simple bean to test bean setting.
 */
public class StudentRegistry {
```

```
  /**
   * The list of students in the registry.
   */
  private static List<Student> students = new ArrayList<Student>();


  /**
   * Add a student to the registry.
   *
   * @param  inStudent The student to add.
   * @return           A string representation of the registry.
   */
  public String addStudent(final Student inStudent) {

    students.add(inStudent);
    return students.toString();

  } // End addStudent().


} // End class.
```

Further, let's say that the `Student` class looks like this:

```
package app;


/**
 * A simple bean to test bean setting.
 */
public class Student {


  /**
   * Student's name.
   */
  private static String name = "";


  /**
   * Student's GPA.
   */
  private static float gpa = 0.0f;


  /**
   * Mutator for name.
   *
```

```
 * @param inName New value for name.
 */
public void setName(final String inName) {

  name = inName;

} // End setName().


/**
 * Accessor for name.
 *
 * @return Value of name.
 */
public String getName() {

  return name;

} // End getName().


/**
 * Mutator for gpa.
 *
 * @param inGpa New value for gpa.
 */
public void setGpa(final float inGpa) {

  gpa = inGpa;

} // End setGpa().


/**
 * Accessor for gpa.
 *
 * @return Value of gpa.
 */
public float getGpa() {

  return gpa;

} // End getGpa().


/**
 * Overriden toString method.
```

```
   *
   * @return A reflexively built string representation of this bean.
   */
  public String toString() {

    String str = null;
    StringBuffer sb = new StringBuffer(1000);
    sb.append("[").append(super.toString()).append("]={");
    boolean firstPropertyDisplayed = false;
    try {
      java.lang.reflect.Field[] fields = this.getClass().getDeclaredFields();
      for (int i = 0; i < fields.length; i++) {
        if (firstPropertyDisplayed) {
          sb.append(", ");
        } else {
          firstPropertyDisplayed = true;
        }
        sb.append(fields[i].getName()).append("=").append(fields[i].get(this));
      }
      sb.append("}");
      str = sb.toString().trim();
    } catch (IllegalAccessException iae) {
      iae.printStackTrace();
    }
    return str;

  } // End toString().


} // End class.
```

Now, how do you go about adding a student to the `StudentRegistry` from JavaScript? As it turns out, it couldn't be easier, as you can see in the `callStudentRegistryAddStudent()` function, which is executed when you click the add student test button:

```
var names = new Array();
names[0] = "Eric Lehnsherr"; names[1] = "Charles Xavier";
names[2] = "Scott Summers"; names[3] = "James Logan";
names[4] = "Bobby Drake"; names[5] = "Katherine Pryde";
names[6] = "Ororo Monroe"; names[7] = "Jean Grey";
names[8] = "Piotr Rasputin"; names[9] = "Hank McCoy";
names[10] = "Warren Worthington"; names[11] = "Kurt Wagner";
names[12] = "Mortimer Toynbee"; names[13] = "Cain Marko";
names[14] = "Yuriko Oyama";
var student = {
```

```
  name : names[studentCount++],
  gpa : Math.round(4.0 * Math.random())
};
if (studentCount > 14) { studentCount = 0; }
StudentRegistry.addStudent(student,
  callStudentRegistryAddStudentCallback);
```

If there were other fields that we are not interested in setting, the field would be left unset in the object. You would need to enable the `bean` or `object` converters for this to work, but once you do, it's automatic. This is a case where you wouldn't need to specify a callback by the way, unless you had a need to be notified when the addition completes.

Extended Data Passing to Callbacks

You probably have noticed that the callback functions receive a single parameter, that being the return data from the remote method invocation. This is oftentimes not enough, and you will want to pass other information. Look and DWR shall provide . . . well, in point of fact, JavaScript itself shall provide!

The answer to this potentially thorny issue is to use a closure.

```
var extraInformation = "Susan Storm";
var callbackProxy = function(methodReturn) {
  realCallback(methodReturn, extraInformation);
};
var realCallback = function(inMethodReturn, extraInformation) {
  alert(inMethodReturn + extraInformation);
}
function callRemoteClassRemoteMethod() {
  RemoteClass.remoteMethod("Reed Richards",
    { callback : callbackProxy });
}
```

The reason this works is that there is no overloading in JavaScript; you can arbitrarily pass parameters to a function. As long as the function is expecting fewer parameters than you pass, nothing will break. Here, when `callRemoteClassRemoteMethod()` is called, as the result of clicking a button, the call is made to `RemoteClass.remoteMethod()`, passing `callbackProxy` as the value of the `callback` member of the metadata object (note that assigning a function to a variable works the same as passing a reference to the function itself—it's just two ways of referencing the same thing). The real callback function named `realCallback()` (as if you couldn't guess) takes two parameters, the return value from the remote method invocation, and some other parameter, some other extra information. The `callbackProxy()` function, however, accepts the single parameter DWR sends it. So, DWR calls `callbackProxy()` when the response returns, which then calls the `realCallback()`, passing it that return value **plus** the extra information. The proxy function will have as part of its execution context that extra information, so it'll be passed along exactly as expected.

A WORD ON CLOSURES

Closures are one of those concepts that confuse the heck out of most developers initially, until it finally just suddenly clicks, and then they see how very useful they are.

A closure can be defined as an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression). Perhaps a simpler explanation is that functions in JavaScript can have inner functions. These inner functions are allowed to access all the local variables of the containing function, and most importantly, even after the containing function returns.

Each time a function is executed in JavaScript, an execution context is created. This is conceptually the environment as it was when the function was entered. That's why the inner function still has access to the outer function's variables even after return: the execution context of the inner function includes the execution context of the outer function.

So, in the previous example, the proxy stub forms an execution context that contains a reference to the `extraInformation` variable. So, when it calls the `realCallback()` function, `extraInformation` is included in the context.

Interacting with DWR on the Server

As I mentioned previously, you will generally want to write your server-side code in such a way that it isn't aware that it's being called from DWR. This will usually not be a problem, but sometimes it may be. With that in mind, there are some facilities DWR makes available to interact with it from the server-side code, and that's what we're going to look at now.

Two classes come to the forefront when working with DWR on the server: `WebContext` and `WebContextFactory`. They are used in tandem like so:

```
org.directwebremoting.WebContext wc =
  org.directwebremoting.WebContextFactory.get();
```

Once you have the `WebContext` (it uses a `ThreadLocal`, so there's always one per request thread), you can gain access to most of the standard servlet objects that you're familiar with, like the following:

```
javax.servlet.http.HttpServletRequest request = wc.getHttpServletRequest();
javax.servlet.http.HttpServletResponse response =
  wc.getHttpServletResponse();
javax.servlet.ServletConfig config = wc.getServletConfig();
javax.servlet.ServletContext context = wc.getServletContext();
javax.servlet.http.HttpSession session = wc.getSession();
```

You can also call `wc.getContainer()`, which returns a `Container` object. This is a very basic IoC (Inversion of Control) container that DWR provides. DWR also provides integration with the very popular Spring IoC container, and in all likelihood you'll want to use that if you're interested in IoC capabilities, as the basic DWR container is just that: very basic.

DON'T REJECT INJECTION!

Inversion of Control, or IoC as it is typically abbreviated, is a technique used in object-oriented programming that decreases the coupling between units of code. IoC is also referred to as Dependency Injection, which describes how it is usually implemented.

Let's say that a class, call it A, depends on another class, B. This is said to be true when any of the following conditions apply:

- A has an instance of B.
- A is a B.
- A depends on some other class C that depends on B, which means A indirectly depends on B.

Note: Just because A depends on B doesn't necessarily imply that B depends on A.

IoC is concerned with how A gets its reference to B. Commonly, class A, if not using IoC, has to instantiate an instance of B itself, or else call some lookup mechanism to get an instance. Class A now effectively has a dependency on the lookup mechanism, and likewise, any class calling class A now has that dependency indirectly.

In IoC, an instance of class B is "injected" into class A, either via constructor or after construction via setter method. The advantage is that dependency is reduced in the code itself, and if the method of getting the instance of class B ever needs to change, class A won't know about the change, nor will any callers of class A. This also lends itself to easier testing since an external test client can easily instantiate the objects a class needs and inject them, regardless of the injection mechanism the code would use for real.

IoC has gained a great deal of attention and affection by many developers, in the Java community especially, thanks to the Spring Framework, which brought the concept into the open more than anything else. Most people view it as reducing code and complexity, which is never a bad thing.

An alternative approach exists to getting these objects that will allow you to decouple the server code entirely from DWR (i.e., no need to deal with `WebContext`). This method is as simple as declaring the object you want in your remote signature:

```
public void serverSideObjects(final String inParam,
  final HttpServletRequest request, final HttpServletResponse response,
  final ServletConfig config, final ServletContext context,
  final HttpSession session) {

  System.out.println("inParam = " + inParam);
  System.out.println("request = " + request);
  System.out.println("response = " + response);
  System.out.println("config = " + config);
  System.out.println("context = " + context);
  System.out.println("session = " + session);

} // End serverSideObjects().
```

When you call this method from JavaScript, you simply do not pass anything in the place of the last five parameters. DWR will, through some unknown incantation of black magic, fill in such parameters with the applicable object. One caveat to this, however, is that your JavaScript calls **must** use the "callback as last parameter" idiom, or the "call data metaobject" idiom; you cannot have the callback function be the first parameter. Why this is so is a question Einstein himself would struggle with, but it's not important, only that you keep that one limitation, such as it is, in mind.

Interacting with other DWR classes on the server is not necessary for most situations. Any exceptions will be covered as necessary throughout the book, but if you find yourself thinking you need other integration points, the DWR Javadoc should be your first and last destination.

DWR Configuration and Other Concepts: The engine.js File

In the firstdwr example app, you'll recall that we had an import of `engine.js`. This is, for all intents and purposes, the heart and soul of DWR on the client side of things. This is the JavaScript code that is responsible for marshaling the call from the dynamically generated proxy stub to the real object on the server.

The `engine.js` code exports a `DWREngine` object. This object allows you to set a number of options to control how DWR works. Table 2-3 shows these options and what they are all about.

Table 2-3. *The Options Available via* `DWREngine`

Option	Description
`async`	This is used to specify whether Ajax calls made via DWR are asynchronous or not. When set to `true`, all calls will block the browser until it returns (this is generally not a good idea!).
`headers`	This allows you to set extra headers to attach to the Ajax calls.
`parameters`	This allows you to set request parameters to go along with the request, accessed on the server via `request.getParameter()` as usual.
`httpMethod`	This sets the HTTP method (generally `GET` or `POST`).
`rpcType`	DWR can use a number of methods to perform Ajax. It can use the typical `XMLHttpRequest` object, or it can use a method based on iFrames, or another method based on dynamically creating `<script>` tags.
`timeout`	This allows you to set an amount of time, in milliseconds, after which a request will be cancelled (as far as the client goes anyway, it simply won't perform an action when the request returns, but the server will still continue processing until completion, whatever that completion may actually be).
`errorHandler`	This allows you to set a JavaScript function to call when any client-side errors occur.
`warningHandler`	This is by default not used, but it allows you to handle, in the same fashion as `errorHandler`, any errors that don't automatically cause the application to blow up. These types of things are usually caused by browser bugs and are "handled," so to speak, by catching them and eating them, in the absence of this being set.

Option	Description
`textHtmlHandler`	This allows you to set JavaScript to execute when a response is received from the server that isn't a proper DWR-created response. This will most usually be the case when a session timeout occurs, but could also be security related, among other causes.
`preHook`	A function to be executed just before the Ajax call is made. This allows you something of an Aspect-Oriented methodology, along with `postHook`.
`postHook`	Similar to `preHook`, except that this will be executed after the call returns, but before the callback is called.
`ordered`	This specifies whether DWR should provide ordered execution of multiple calls. This is accomplished by only sending a single request at a time, queuing up the rest until each returns. Note that this will slow down your application, and worse still, may result in a hung application if something goes wrong along the way. It is generally better to redesign your application to avoid the need for ordered execution in the first place than to use this option.
`pollType`	When using reverse Ajax (a topic covered later), this specifies whether to use the `XMLHttpRequest` method or iFrame method.
`reverseAjax`	This specifies whether DWR should be looking for inbound calls with each outbound request (again, covered later with the reverse Ajax coverage).

Each of these options can be set in three ways. First, they can be set globally, so that they affect every call DWR makes. To do this, you call a setter method. For instance, to set the timeout option globally, call `DWREngine.setTimeout()`. Second, they can be set at the batch level (by passing them as parameters to the `endBatch()` function). Third, they can be set at a call level, by sending them as parameters to the method itself, much like you pass the callback function to it. Further, if you have a global setting, the setting at the call level will override it, quite like you'd expect it to work. Two exceptions are the `preHook` and `postHook` options, which are additive, meaning you can have multiples per call or per batch, and they will be called in the order you'd expect, meaning those set globally first, then those at the batch level, then those at the call level.

One other thing of note is that certain options, namely the `ordered`, `pollType`, and `reverseAjax` options, cannot be applied to an individual call, whereas the other options can.

We'll see all of this in action as we work through the applications, so don't fret if it seems somewhat confusing right now.

I just had to quote from the DWR documentation here, because I couldn't write something this funny myself! In discussing how all of this might seem a little complex, it states:

> "If all of this sounds confusing then don't worry. DWR is designed to do what you expect, so it should not be complex."

Do I really need to point out the irony of that statement? Isn't it something like the piece of paper that on one side says "The sentence on the other side of this paper is true," and on the other side it says "The sentence on the other side of this paper is false"? But, it happens to be a true statement: DWR generally works as you'd expect, so even if the options seem a bit confusing at times, it works out logically in practice.

Whoa, what's that batching thing I just mentioned? Let's get into that now.

Call Batching

It is possible to make a collection of calls, or a batch, at one time. This cuts down on the network traffic required, thereby making your application more responsive due to lower overall latency. To perform batching, you simple do this:

```
DWREngine.beginBatch();
  BatchCallClass.method1(testCallBatchingCallback1);
  BatchCallClass.method2(testCallBatchingCallback2);
  MathDelegate.multiply(8, 4, testCallBatchingCallback3);
DWREngine.endBatch();
```

When the `endBatch()` call is made, DWR takes all the method calls you've made and sends them all as one Ajax request. DWR will then take care of calling the callbacks for all the batched requests. It is important to remember to call `endBatch()` at some point; otherwise DWR will simply batch all subsequent calls forever, and you'll be left wondering why your application never sends any requests, much like how your friends never return your calls when you need help moving!

In the seconddwr example webapp, I've added a very simple filter that displays the message "Request came in" to the console and log file whenever a request mapped to the path `/dwr/` comes in. When you click the button to test call batching, note that only a single message is printed, indicating a single request. This is the benefit of call batching: it's not simply DWR making three different calls and then calling the callbacks, it's literally reducing it all to one network request, which is a huge benefit in many cases where you know your code is structured in such a way that batching makes sense.

A Quick Look at util.js, the DWR Utility Package

One last thing that we should look at is `util.js`, which is a collection of useful functions that DWR offers for updating the page dynamically using data, such as that returned from a remote call. The nice thing about `util.js` is that it isn't tied to DWR specifically (with one exception, that being the `useLoadingMessage()` function), so whether you use DWR or not, you can extract `util.js` out of the DWR example WAR file and use it separately (note that here, "extract" means you access the URL for it, which is `<CONTEXT>/dwr`, assuming you've configured the `DWRServlet` as in our firstdwr example app, and save what the server returns, which is `util.js`).

So, what all does `util.js` offer? Let's have a look:

- `addOptions()`: This is a function that allows you to add elements to lists (ordered and unordered) as well as `<select>` elements. It allows you to pass it a simple array of strings, or an array of objects, a single object, or a map of objects.
- `addRows()`: This function allows you to dynamically add rows to a table. Among other options, it allows you to specify a function to call while creating each row and/or each cell, so you can tailor the addition to your needs.

- `byId()`: This is more or less equivalent to the ubiquitous `document.getElementById()` method. Shorter to type though, so right on!
- `getText()`: This function allows you to get the text (not the value) of an `<option>` element (child of `<select>`).
- `getValue()`: A nice function that allows you to get the value of virtually any HTML element without having to consider whether it's a text box, a `<div>`, or something else.
- `getValues()`: This is like `getValue()` except that you can specify a bunch of HTML element IDs, and the value of each will be retrieved.
- `onReturn()`: This adds an event handler to an element that handles pressing the Return key. This is generally used in the context of forms. This function deals with the differences in how the browsers handle the Return keypress.
- `removeAllOptions()`: This allows you to remove all the elements from a list or `<select>` element.
- `removeAllRows()`: A function that removes all rows from a specified table.
- `selectRange()`: This function gives you the ability to select a portion of text in a text box, without having to worry about the many differences between browsers for doing so.
- `setValue()`: This is akin to the `getValue()` function, except that it of course sets the value of the specified element. Once again, it can be used with nearly any HTML element, regardless of type.
- `setValues()`: Similar to `setValue()`, but like `getValues()`, it allows for dealing with multiple elements at once.
- `toDescriptiveString()`: This in short is a better version of the default `toString()` method, providing a more, well, **descriptive** string representation of an object.
- `useLoadingMessage()`: This allows you to have a loading message appear when a remote call is in progress, like Gmail famously does. This is the one function that is tied to DWR. It should also be noted that the documentation states this may be deprecated at some point in the future due to a number of limitations, so you may want to use it with caution.

All of the functions have a prefix of `dwr.util`, meaning that you form their complete name by appending that to the function name you want to call. So, `setValue()`, for instance, would be `dwr.util.dwrValue()`. You can also use `DWRUtil` instead of `dwr.util`, which is the way it was set up prior to DWR 2.0, but that may be deprecated in the future, so it would be better to not use it at all. However, if you see any example code floating around the Internet that uses `DWRUtil`, it's referring to the same thing as `dwr.util`.

If it seems like there wasn't much detail to these descriptions, that's by design: we'll be looking at each of these throughout the book as their usage comes up. This is just a quick overview of what is available. Keep reading, all shall be revealed!

Summary

I'd like to paraphrase the immortal words of Keanu Reeves in *The Matrix*:

Whoa. I know DWR!

Yes, I know, it was actually two separate lines from two separate scenes, but allow me a little creative license, OK?

In this chapter, we got our first real look at DWR. We talked about what it is, what it does, how it does it, and why it's cool. We discussed why you should (or shouldn't) use DWR. We then moved on to setting up the development environment in preparation for playing with DWR. Then, we moved on to actually playing around with it by building a simple example webapp. Along the way, we saw how to configure DWR (at least the basics) and looked at some of the options it offers. Finally, we took a quick look at the utility functions DWR provides on the client for working with our Ajax responses.

In the next chapter, we'll take a look at some more advanced topics, including integration with other popular frameworks and something called reverse Ajax.

CHAPTER 3

Advanced DWR

In the previous chapter, we dove head-first into DWR. We saw how to configure it in an application and how to interact with it on the client as well as the server, and we basically nailed down most of the basics that you'll need to use it day in and day out. In this chapter, we'll take a few more steps along the path of understanding, learning about some of the more "advanced" topics DWR has to offer. I put "advanced" in quotes because nothing in this chapter is any more complex from a technical standpoint than what we've seen already, by and large, but these are the things that you may not have to deal with every time you use DWR. We'll take a look at security, error handling, accessing other URLs, reverse Ajax, integration with other frameworks, and Java 5 annotation support in DWR.

Locking the Doors: Security in DWR

It's a well-known fact that all manner of nastiness awaits your lowly webapp the second you expose it to the Internet. There are legions of hackers ready to attack you in every way imaginable, and then a few more. They want nothing more than to steal your money, trash your goods, destroy your reputation, overfeed your goldfish, and kick over your trash the day after you finally get around to cleaning out the garage. If left to their own devices, they will own your site, using it to stage zombie attacks against endangered bird species that only the hallowed DMCA[1] can defend against. "Weird Al"[2] got it right!

(I have been known to exaggerate for effect from time to time.)

That was all obviously on the facetious side, to say the least. But, the point that security is an important consideration in any web development effort is absolutely serious and true. This is no less true in the world of Ajax, and therefore in the world of DWR as well.

---

1. The DMCA, or Digital Millennium Copyright Act, is a law in the United States that criminalizes activities such as creating software or devices whose purpose is to circumvent access control technologies (commonly referred to as DRM, or Digital Rights Management). It also expands penalties for copyright infringement on the Internet. The DMCA is one of the most reviled pieces of legislature in the whole of human history because many see it as being too far reaching, unfair, and even a form of censorship (owing to the ease with which copyright holders can have content removed from web sites, for example, even when infringement is legally questionable). Some even believe it to be an illegal measure itself. Unfortunately, no court case to date has successfully challenged it in any meaningful way.
2. "Weird Al" Yankovic released a song called "Virus Alert" on his 2006 album *Straight Outta Lynwood*. It's a song about discovering a virus on your computer and, shall we say, blowing it **way** out of proportion! For your pleasure and diversion: `www.answers.com/topic/virus-alert`.

However, where many other Ajax tools frankly leave you out in the cold as far as security goes (see Figure 3-1), letting you fend for yourself, DWR offers some capabilities in this area that you may well find to be all you need.

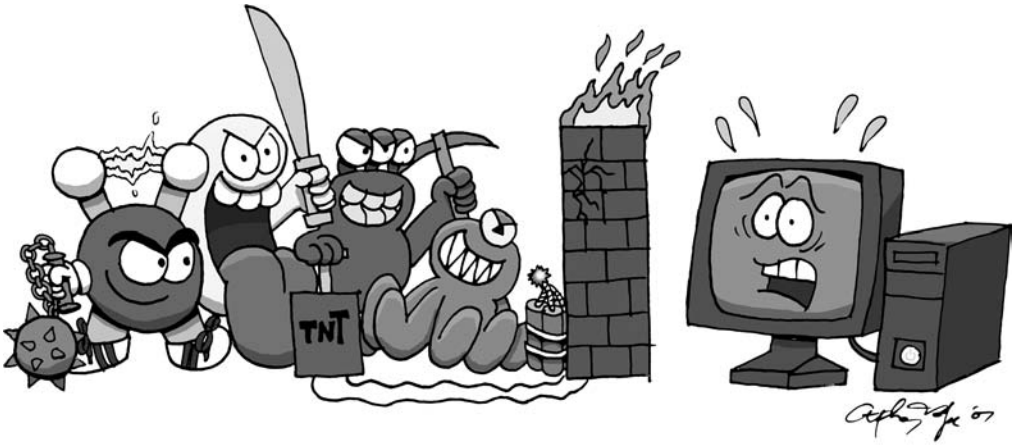


Figure 3-1. *Security threats, while perhaps not as bad as made out to be in some movies, are very real and not typically as cute as in this illustration!*

DWR takes a multitiered approach to security, but it always has at least one mechanism active at all times. Let's take a look at that mechanism first.

Deny by Default

Without your having to do a thing, DWR automatically takes a "deny by default" approach. This means that by default, DWR will not allow remoting of any class. Recall that for every class you wish to have remote access to, you must add a `<create>` element in the config file. As if that wasn't enough, remember that it's actually nested under an `<allow>` element, which should be a big enough clue to knock you unconscious all by itself! By adding such an entry, you are telling DWR that it is OK to make remote calls on that class. Failing such an entry in the config file, however, remote calls to a class cannot be made.

The astute reader will no doubt recognize that this alone actually opens up a potential security hole: by default, **all** methods of an allowed class will be accessible via remote call. In general, this is probably OK because you will likely organize your code such that a class will always contain what we could call "remote-safe" methods. Designing a class that had some methods that were safe for remoting and some that weren't probably wouldn't be a great design. Still, there may be justification for doing so, and in that case DWR offers some further configuration flexibility.

Consider the configuration shown in Listing 3-1.

Listing 3-1. *Sample* `dwr.xml` *Configuration*

```
<?xml version="1.0" encoding="UTF-8"?>


<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">


<dwr>
  <allow>
    <create creator="new" javascript="MathDelegate">
      <param name="class" value="app.MathDelegate" />
    </create>
  </allow>
</dwr>
```

This is the `dwr.xml` from the firstdwr example app in Chapter 2. In this case, we want all the methods of the `MathDelegate` class to be callable remotely. However, let's say for the sake of argument that we didn't want to allow the `divide()` method to be callable. We can simply add an `<exclude>` element, as was mentioned briefly in the previous chapter. As it turns out, there is an `<exclude>` as well as an `<include>` element that is a child of the `<create>` element. These two elements are mutually exclusive, that is, you either specify a list of methods that are remotable by specifying an `<include>` element (anything not listed **will not** be accessible) or you specify a list of methods that are not remotable, by specifying an `<exclude>` element (anything not listed **will** be accessible). In Listing 3-2, you can see the updated `dwr.xml` file that excludes the `divide()` method.

Listing 3-2. *Sample* `dwr.xml` *Configuration with* `divide()` *Excluded*

```
<?xml version="1.0" encoding="UTF-8"?>


<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">


<dwr>
  <allow>
    <create creator="new" javascript="MathDelegate">
      <param name="class" value="app.MathDelegate" />
      <exclude method="divide" />
    </create>
  </allow>
</dwr>
```

Both <include> and <exclude> accept a comma-separated list as the value of their method attributes, so you can specify as many methods in either as you like.

Note that DWR will **never** allow remoting of its own internal classes, thereby reducing the possibility of an attacker messing around with DWR internals and circumventing the rest of your application's security.

J2EE Security and DWR

If you nourish a healthy sense of paranoia when it comes to securing your webapps, you can take the next step and integrate J2EE security with DWR. With this capability, you can specify that only certain security roles may access certain remotable classes, or that only certain roles can access certain methods of a remotable object.

WHAT IS J2EE SECURITY?

The term J2EE security typically refers to the security model that the J2EE platform, and container that implements it, usually provide. This mechanism defines roles, which users are then assigned to. Resources are said to be "constrained" such that only certain roles can access them, and therefore only certain users can. This mechanism is managed by the container, not your application code, alleviating the need to write all the boilerplate security code yourself, which tends to be less secure and more difficult to audit. All of this capability comes in a declarative form, meaning there's usually nothing more to do than create appropriate entries in a configuration file (or files, depending on your container). While there are programmatic hooks into the mechanism as well, they aren't usually required to actually implement the security, just to enhance your application with it sometimes.

Securing the DWRServlet

The first option is to secure the DWRServlet itself using J2EE role-based security. To do so, we have to introduce another configuration option: the ability to have more than one dwr.xml file.

Recall that in web.xml, you write something like this:

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
</servlet>
```

Did you notice that dwr.xml is not specified anywhere? That's because by default, DWR looks for a file named dwr.xml in WEB-INF of your webapp, so you don't have to do anything more. However, if you want to place the config file elsewhere, you simply add an init parameter to the servlet like so:

```
<init-param>
  <param-name>config</param-name>
  <param-value>configFiles/dwrConfig.xml</param-value>
</init-param>
```

DWR will now load the file `dwrConfig.xml` from the directory `configFiles`, relative to the root of the webapp.

Another handy trick is that you can specify multiple configuration files. To do this, you simply add more init parameters with a name in the form configXXXX, where XXXX can be anything. The init parameters must all have different names, so the XXXX portion simply serves to make them unique. For example, you could do the following:

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
<init-param>
  <param-name>config1234</param-name>
  <param-value>configFiles/dwrConfig1234.xml</param-value>
</init-param>
<init-param>
  <param-name>config5678</param-name>
  <param-value>configFiles/dwrConfig5678.xml</param-value>
</init-param>
</servlet>
```

This too is a handy trick because it allows you to organize your configuration into logical groups, e.g., one config file per package or some other structure you deem reasonable. DWR will load all the config files and combine them into one master configuration—no extra effort on your part required!

How does this factor into security you ask? Well, something else you can do is define multiple instances of `DWRServlet`:

```
<servlet>
  <servlet-name>dwr-user-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
    <param-name>config-user</param-name>
    <param-value>WEB-INF/dwr-user.xml</param-value>
  </init-param>
</servlet>
<servlet>
  <servlet-name>dwr-admin-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
    <param-name>config-admin</param-name>
    <param-value>WEB-INF/dwr-admin.xml</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>dwr-admin-invoker</servlet-name>
  <url-pattern>/dwradmin/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
```

```
  <servlet-name>dwr-user-invoker</servlet-name>
  <url-pattern>/dwruser/*</url-pattern>
</servlet-mapping>
```

What we have here is essentially two logical groupings: plain users and administrative users. We have a different config file for each, and more importantly, a different URL mapping for each. The reason this is important is because we can now leverage J2EE security to secure these servlets by role. For example:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>dwr-admin-collection</web-resource-name>
    <url-pattern>/dwradmin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>dwr-user-collection</web-resource-name>
    <url-pattern>/dwruser/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>user</role-name>
  </auth-constraint>
</security-constraint>
```

Now, only users in the admin role will be able to access the classes defined in the `dwr-admin.xml` file, and accessed via the `/dwradmin/*` URL pattern. Likewise, only users in the `user` role will be able to access the classes defined in the `dwr-user.xml` file, and accessed via the `/dwruser/*` URL pattern. We've effectively secured our remotable objects using standard-based container-managed security without breaking a sweat!

Securing Individual Methods

The other possibility is to secure individual methods on a given remotable class. This is perhaps even easier to configure. All you need to do is ensure the roles are defined in `web.xml`, and then in the DWR configuration file you add an `<auth>` element to your `<create>` element for the class you're securing. Here's what it looks like:

```
<create creator="new" javascript="RemotableClass">
  <param name="class" value="com.myapp.RemotableClass" />
  <auth method="delete" role="admin" />
</create>
```

Here, we are specifying that the `delete()` method of the `RemotableClass` class can only be called by users in the `admin` role. Note that all other methods will be unprotected, as is the default with DWR. This fine-grained control gives you a lot of flexibility in how you apply container-managed security, and therefore how you design the classes you wish to remote.

DWR's J2EE security support, both types, will be demonstrated in the Chapter 5 project, so don't worry, you'll get to see this all in action just a short time from now. (DWR actually makes writing about it hard at times because it makes things so easy you find it hard to put enough words on the page . . . I feel like there should be more here, but that's about all there is to it, really!)

When Perfection Is Elusive: Error Handling in DWR Applications

It has been said that those who write code for a living are in the business of using chaos to their advantage to only allow the user enough glimpses into the coder's inability to write solid code via errors about 10 percent of the time. The other 90 percent of the time it'll be abundantly clear to the user.

OK, no one ever said that, but someone probably should have before now.

The simple fact is that if you develop software for a living, you deal with errors. You deal with exceptional situations that inevitably arise at the most inopportune time during the execution of your code. You do your best to handle them gracefully, and that requires forethought and extra effort put in.

Bugs and runtime problems lurk around every corner (see Figure 3-2), eating away at your carefully crafted code! This is no less true with DWR than anything else.

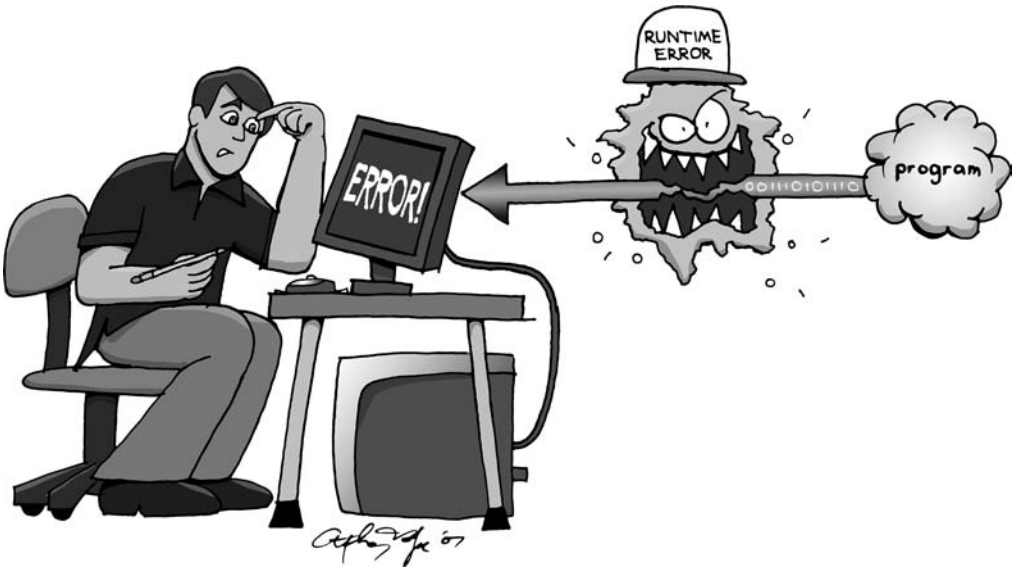


Figure 3-2. *Bugs and runtime problems lurk around every corner, eating away at your carefully crafted code!*

Fortunately, DWR doesn't completely leave you out in the cold. In fact, it has a fairly robust system of error/exception handling that lets you deal with things in the most appropriate manner, and as the situation warrants. It does this by allowing you to set up handlers for

the various types of exceptional situations that can arise, and it allows you to do so on a number of levels. First, let's look at the way DWR categorizes these exceptional situations.

Handling Warnings

First of all, DWR conceptually breaks what you can generically refer to as exceptional states into three categories, the first of which is warnings.

Warnings are things that may or may not represent a condition we care about. There are situations where something looks to have broken, but really it hasn't. One such situation pointed out in the DWR documentation can arise in Firefox. Under some conditions, it is possible for the response to an Ajax request to come back after the `XMLHttpRequest` object that made the request has already been destroyed. This can occur if you fire the request and then transition to a new page before the response comes back, for instance. In such a case, you almost certainly can do nothing to remedy the situation and likely wouldn't even want or need to code for it.

For this reason, by default, DWR supplies no handler for warning conditions, and they are effectively ignored.

Handling Errors

Next up the severity ladder is errors. Errors are situations where something goes wrong, and DWR can definitively determine what happened. These are errors that are handled on the client side of the house by the way, not the server side. One example might be if the application server shuts down in the middle of servicing a DWR request. This will result in an HTTP error, and DWR can handle this because it can see the error code and act accordingly.

Handling Exceptions

The final category is exceptions. These are errors that occur on the server side, and which propagate out to the client side. These are actually treated like errors when they hit the client, but they really are conceptually a whole different category. However, they are treated like errors even to the extent of being dealt with by the same handler, so in practice you may find little difference.

Edge Cases: Improper Responses

There exists one other category that kind of stands on its own, but before you can understand it you need to understand what DWR is passing back and forth between the client and server.

When a request is made, it's a fairly typical HTTP request. You can see what a DWR request looks like by using the handy Firebug extension to Firefox, which results in what you see in Figure 3-3.

As you can see, it's a pretty typical HTTP `POST` operation. You can see the two numbers I entered being passed in, along with the name of the method being called on the remote object (`add()` in this case), the name of the JavaScript object (`MathDelegate`), and some other information DWR uses internally to do its work.

Figure 3-3. *Anatomy of a DWR Ajax request*

We can see the response just as easily with Firebug, and it's probably close to what you'd expect, as shown in Figure 3-4.

Figure 3-4. *Anatomy of a DWR Ajax response*

What's passed back is literally a JavaScript call to some DWR client code. This code will then take care of calling the callback function you specified, after doing some of its own housecleaning presumably. You can see how it passes the original parameters into that function, as well as the result of the remote method invocation. You will also note some lines above that, which are just JavaScript comments. These lines are pretty important, as we'll see shortly.

Before we look at that though, we may as well look at one more response from DWR, and that's the case of a bean being passed back. Figure 3-5 shows just such a response.

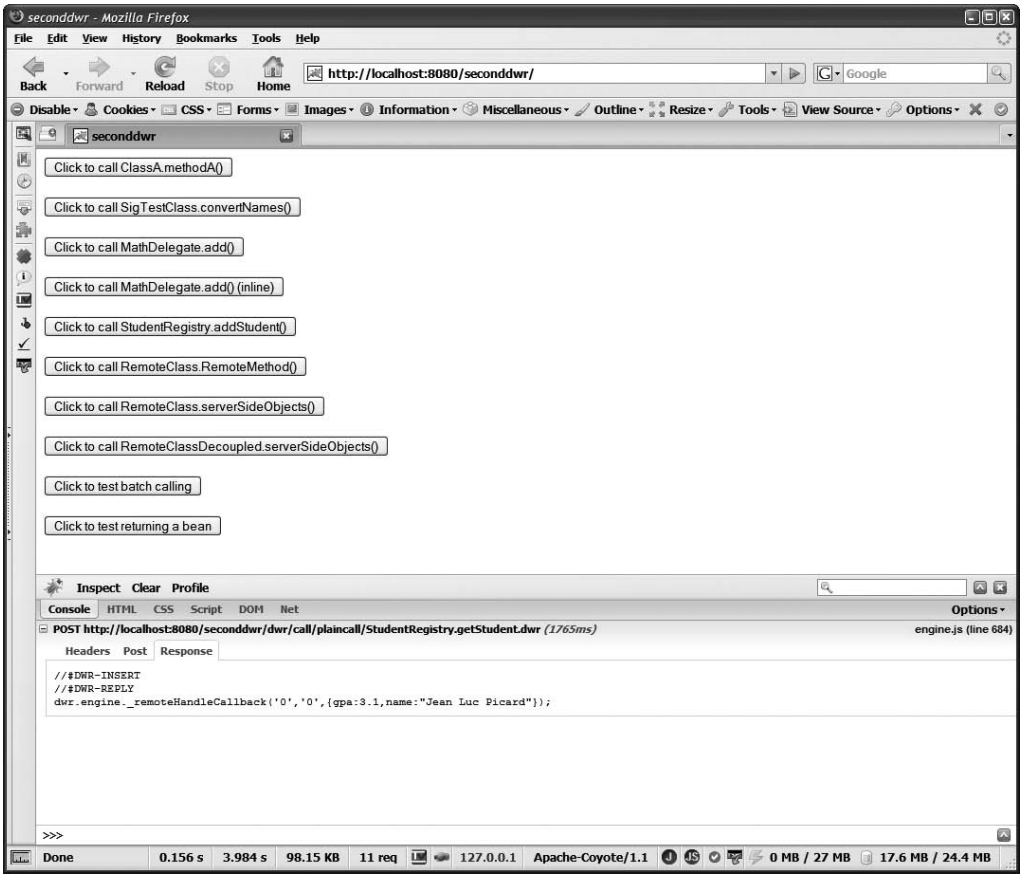


Figure 3-5. *An object being returned by a DWR Ajax request*

This response was generated using a slightly modified version of the seconddwr application from Chapter 2. All I did was add a `getStudent()` method to the `StudentRegistry` class that returned a populated `Student` object. I then added a button to the JSP to call that method. I suggest taking a moment and using this as a quick little exercise to help get these concepts embedded in your brain. Doing that same modification (because it's not included in the downloaded code) would be a worthwhile endeavor I think.

As you can see, it isn't a whole lot different really, but the important point to take away is that the bean that is returned is sent in the form of JSON. This is how a great deal of data interchange between client and server is done in the Ajax world, and DWR is no exception.

WHAT'S THIS JSON YOU SPEAK OF?

JSON, which stands for JavaScript Object Notation, is a simple data interchange format that is roughly similar to array notation in most C-like languages. JSON is technically a subset of the object literal notation used in JavaScript to define objects.

A quick and simple example of JSON is this: `{ firstName:"frank", lastName:"Zammetti" }`. If you were to execute `var p = eval(json);` where `json` is the example JSON shown here, what you would have is an object with the fields `firstName` and `lastName` present and pointed to by the variable `p` such that doing `alert(p.firstName);` would result in a pop-up with the text "frank" in it.

JSON has quickly become very popular in Ajax development and is at this point perhaps the de facto standard for client-to-server communication in webapps today. Its primary benefits are simplicity in terms of creation and consuming, lightweight (as compared to the typical alternatives such as XML), and the fact that it's still relatively human-readable. It, like XML, is essentially self-describing, assuming the person or system generating it isn't brain-dead.

It allows for common structures needed such as arrays (lists) and maps, as well as arbitrarily complex nesting depths to allow for object hierarchies.

While it is popular in Ajax development, it is by no means limited to that domain. In fact, JSON, since it's just a plain text format, can readily be used in many circumstances where you have data interchange needs.

Now, let's get back to those JavaScript comments in the response. These are the key to how DWR handles this final "fringe" category of failures. Imagine if you will what would happen if a user is on a web site that uses DWR and he or she is sitting idle for a few hours (that user shouldn't have eaten that gas station tuna fish sandwich for lunch!), and now he or she hits a button that fires off an Ajax request. Assuming the application was written with any semblance of normalcy, the session will time out. As I'm sure you know, that timeout will be handled by the container (or web server if there's one in front of it) long before DWR or the remote object that's being called is hit. So, what's returned to the client-side DWR code making the call? Well, it will depend on the server configuration, but clearly it won't be a proper DWR request.

DWR will determine this, primarily, by looking for those comment lines in the response. If they aren't present, this is considered an "improper" response from DWR's point of view. Like all other conditions so far discussed, you have the ability to handle this situation if you wish. The alternative is to let DWR silently eat them, which probably isn't what you want for a robust application.

The Mechanics of Handling Exceptional Situations

So, now that you understand the various scenarios that can lead to warnings, errors, exceptions, and improper responses, the next logical question to answer is how do you actually code for them? As with most things in DWR land, it's very simple. In fact, the very simplest form (aside from doing nothing and letting the DWR defaults be in effect) is this:

```
dwr.engine.setErrorHandler(someFunction);
```

For all errors that occur, `someFunction()` will be called, and you can take whatever action you deem necessary at that time.

Similarly, if you wish to handle warnings, you can do the following:

```
dwr.engine.setWarningHandler(someFunction);
```

Finally, to handle those improper responses that might possibly come back from the server, you need to set up what's called a `textHtml` handler, like so:

```
dwr.engine.setTextHtmlHandler(someFunction);
```

Now, you're probably wondering about handling exceptions, and of course you can do that too, but there is a difference. While you can handle errors, warnings, and improper responses at a global level, you cannot do so for exceptions. You must handle them on a per-call basis, which leads to the fact that you actually can handle **any** of these types on a per-call basis (or on a per-batch basis, except for exceptions, which are **always** on a per-call basis, even when a call is made as part of a batch). All you need to do is make sure to use the metadata object call mechanism and specify the error/exception/warning/improper response callback. Here's an example:

```
MyRemoteClass.someMethod(param1, param2, {
  callback : function(response) { /*Do work*/ },
  errorHandler : function(errMessage, exception) { /*Do work*/ }
});
```

Again, while exceptions cannot be handled at the batch level, all other types can be, and the syntax for that is very similar to the per-call basis:

```
dwr.engine.beginBatch();
MyRemoteClass.someMethod(param1, param2, function(response) { /*Do work*/ });
// More calls
dwr.engine.endBatch({
  errorHandler : function(errMessage, exception) { /*Do work*/ }
});
```

Another Word on Exceptions

One final thing to discuss about exceptions is how they are marshaled from the server. In short, DWR takes the same "deny by default" tact as it does with remoting of classes. However, this would immediately lead to the conclusion that when an exception occurs, you'd in fact get **no** response back, not a meaningful one anyway, and that's clearly not the right answer.

So, by default, DWR will use what it calls the `MinimalistExceptionConverter`. This will result in a response that DWR recognizes as an error, but it will contain no information about the nature of the error. This is done for security reasons (imagine what a decent hacker could learn about your application just by the information contained in the error messages if he or she continually caused errors on purpose!).

However, you may well want to have more information than this, especially during development. So, you can configure a converter in `dwr.xml` such as

```
<convert match="java.lang.Exception" converter="exception" />
```

Now, any exception matching this (any descendant of `java.lang.exception` in this case) will result in a string of JSON being returned such as this:

```
{ javaClassName : 'java.io.IOException',
  lineNumber : 109,
  publicId : 'pid',
  message : 'Couldn't read file' }
```

That's clearly a lot more useful to you as a developer! However, you may want something in between because in a production environment you may well want more information than the default, but not quite as much as in development. Fortunately, the converter allows you to have more fine-grained control over what information is presented back to the caller, and again, it's nothing more than some added configuration:

```
<convert match="java.lang.Exception" converter="exception">
  <param name="include" value="message,lineNumber" />
</convert>
```

One other thought factors into security here, and that's stack trace elements. Returning stack traces to users, aside from being bad in terms of usability, is a potential security risk. So, by default, DWR won't include stack trace elements. If you'd like to see that, you need to allow DWR to marshal them as well by adding this configuration element:

```
<convert match="java.lang.StackTraceElement" converter="bean" />
```

As with any time you decide to dump a stack trace, realize that it can be rather long and verbose, so it's definitely something you want to use with caution.

Help from Elsewhere: Accessing Other URLs

Let's say you have a class that you are calling via DWR, and in this class you need the services of another part of the application. However, this part of the application isn't exposed via DWR. For instance, you need to show a particular entry form to the user, so all you really want to do is insert some markup into a `<div>` on the page. However, the form is created dynamically using Velocity templates. You may think DWR can't really help you here, but in fact it can.

DWR gives you the ability to fetch content from a URL and return it as the result of a method execution. For instance, examine the class in Listing 3-3.

Listing 3-3. *Reading from Another URL and Returning the Result*

```
package app;


import java.io.IOException;
import javax.servlet.ServletException;
import org.directwebremoting.WebContextFactory;
```

```
/**
 * A class to read from a URL and return the contents.
 */
public class URLReader {


  /**
    * Read a URL and return its contents.
    *
    * @return                  The contents of the URL.
    * @throws ServletException If anything goes wrong.
    * @throws IOException      If anything goes wrong.
    */
  public String read() throws ServletException, IOException {

    return WebContextFactory.get().forwardToString("/another.jsp");

  } // End read().


} // End class.
```

Now, examine the call to this class:

```
function readURL() {
  URLReader.read(
    {
      callback : function(inResp) {
        document.getElementById("divURL").innerHTML = inResp;
      }
    }
  );
}
```

So, what happens is this: the `read()` method of the `URLReader` class is called. It uses the standard servlet API forward mechanism under the covers to get the contents of the specified URL, which in this case is another JSP within the same webapp. That content is then returned as a string as the result of the `read()` method's execution. This content is returned to the client, which then inserts it into the `divURL <div>`.

You don't need to type this in yourself: included in the download source code bundle is an application named `anotherurl` where this code is taken from. Again, laziness has its virtues!

This is a very handy function when you need to return markup. There's no need to write code in classes to generate it, unless you're into that sort of thing, but it pretty quickly leads to unmaintainable, ugly code, so it's better to avoid it.

The bad news is that this is subject to the same rules and limitations as the standard servlet API forwarding mechanism, so you can't use this to fetch remote content, unfortunately. But, it solves the problem of how to generate markup very well, so you should keep it in mind for sure.

Turning the Tables: Reverse Ajax

Just when you start getting your brain wrapped around this whole Ajax thing, they go and change the rules! That's exactly what reverse Ajax is. Figure 3-6 shows reverse Ajax graphically, a nice summation of how it works (it in fact kind of jumps the gun a bit by giving you an idea of what the code looks like that makes it work . . . we'll be getting to that after some discussion of reverse Ajax).

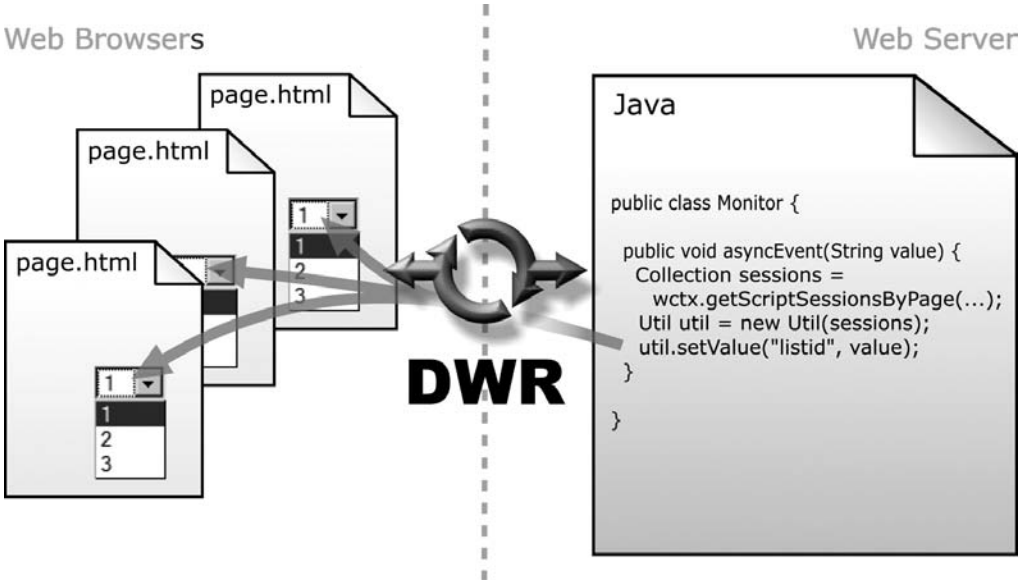


Figure 3-6. *Graphical overview of reverse Ajax in DWR (used with permission of Joe Walker, DWR creator)*

Reverse Ajax is something fairly new, coming to prominence only really in the past few months. The basic concept is that instead of the client having to pull information from the server, the server can push information directly to the client. This is all in an effort to overcome the one limitation of Ajax that persists from the "classic" web model, namely that real-time information is still not technically possible because there is still a requirement that the client has to contact the server and ask "Hey, dummy, has anything changed?" and then update the page (or some section thereof) if anything has. While you can do this quickly enough to make it **seem** it's happening in real time, it really isn't. What's needed is a way for the server to contact all the browsers viewing its contents to announce when changes have occurred.

Reverse Ajax is a way to make that dream a reality. Like Ajax itself, it's not a specific technology but a technique to use existing technologies in some unusual ways.

As you will no doubt recall, the classic Web, that is, non-Ajax applications, has a specific flow of events that characterizes it. Simply stated, a user action on the client results in a request to the server, then some processing occurs on the server, and a response is returned to the client, the response being an entirely new UI view. This continues ad infinitum, until the user decides to leave the web site. In Figure 3-7 you can see a graphical representation of the sequence of events.

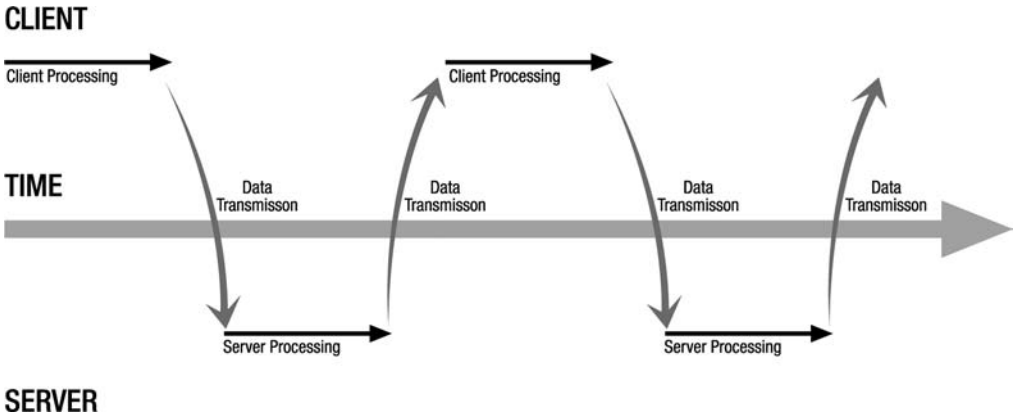


Figure 3-7. *Basic sequence of events for the classic Web*

An Ajax-based application has a slightly different sequence, and Figure 3-8 shows that. There, we can see that some user activity results in a call to some client-side Ajax engine, whether that's our own JavaScript code or some library. This engine makes a request to the server, which does its processing as in the non-Ajax model, then returns the response. The response is handled initially by the Ajax engine, which then typically calls on some client code to update the page. Once again, this sequence continues indefinitely until the user navigates away from the page.

Reverse Ajax presents yet another sequence of events, but it's one based on the Ajax diagram in Figure 3-8, in which there is a two-way flow of information, the usual client-initiated flow and the new server-initiated flow (we'll see some diagrams in a moment, I know you're just awaiting the pretty pictures!).

Now, here's the thing: reverse Ajax is essentially an illusion! To be sure, it's a well-done illusion, but an illusion nonetheless. Within the confines of current HTTP technology, there is no way to truly push information from a server to a client because the protocol is fundamentally stateless, and once the client-initiated connection is broken, the server no longer knows about its clients. There is no notion of an "always-on" connection from client to server, regardless of who may have initiated it.

So, you can't have true push technology, but you can actually emulate it surprisingly well!

There are three techniques at present for performing this magic trick, and DWR supports all three. In addition, two of them fall in the category of "active" reverse Ajax, while the third is considered "inactive" reverse Ajax. Let's take a look at each and what DWR has to offer.

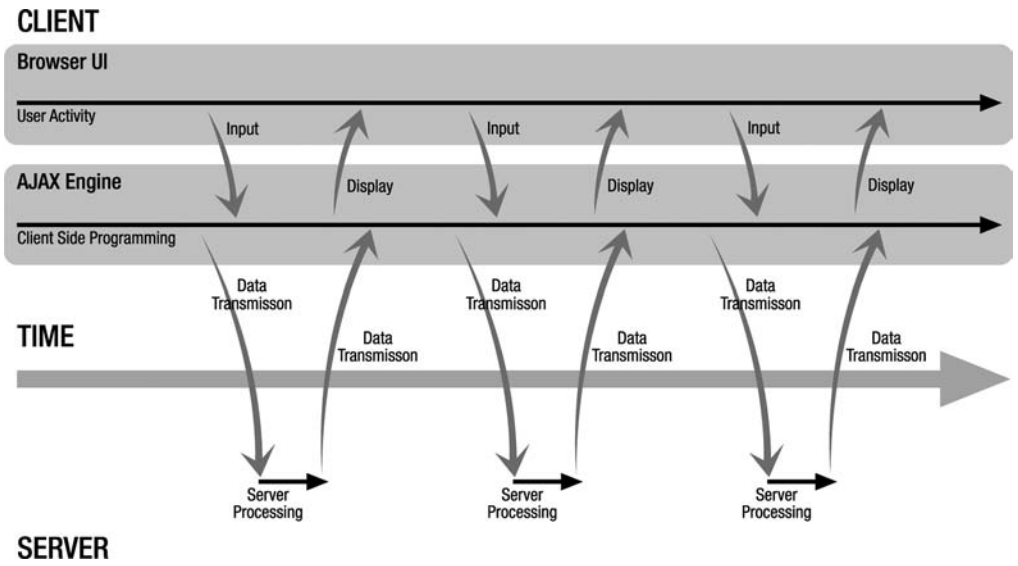


Figure 3-8. *The basic sequence of events in the Ajax world*

Polling

The polling technique is, strictly speaking, not specific to Ajax and has been around for quite some time longer. This is the simplest form of "server push," and you are no doubt already aware of it, even if the term is unfamiliar.

Image a plain old web page, no Ajax or anything special. Imagine you use a `<meta>` refresh tag to continually update the page every few seconds. This is polling in action! The client is continually polling the server for changes and displaying the new information (technically it displays whatever the server reports as the current information, which may or may not be new as compared to what the user was seeing before the last poll event, but the appearance is that new information is displayed while unchanged information is still displayed unchanged).

You can do the same thing with some simple JavaScript on a page that continually refreshes the page. It's all the same thing, and it's all polling.

In the Ajax world, the flow of events is slightly more complex, but still essentially the same, as you can see in Figure 3-9.

In this diagram, you can see the first three poll events occur without any user intervention; the polling client, or Ajax engine, is initiating the request independent of anything the user may be doing. Also note that those first three poll events do not make calls to the browser UI when the response is received. This indicates cases where no update was required because no new data was present.

This is one of the two active reverse Ajax methods because there is an automatic action of some sort involved in getting updates from the server to the client. The other active method is called Comet.

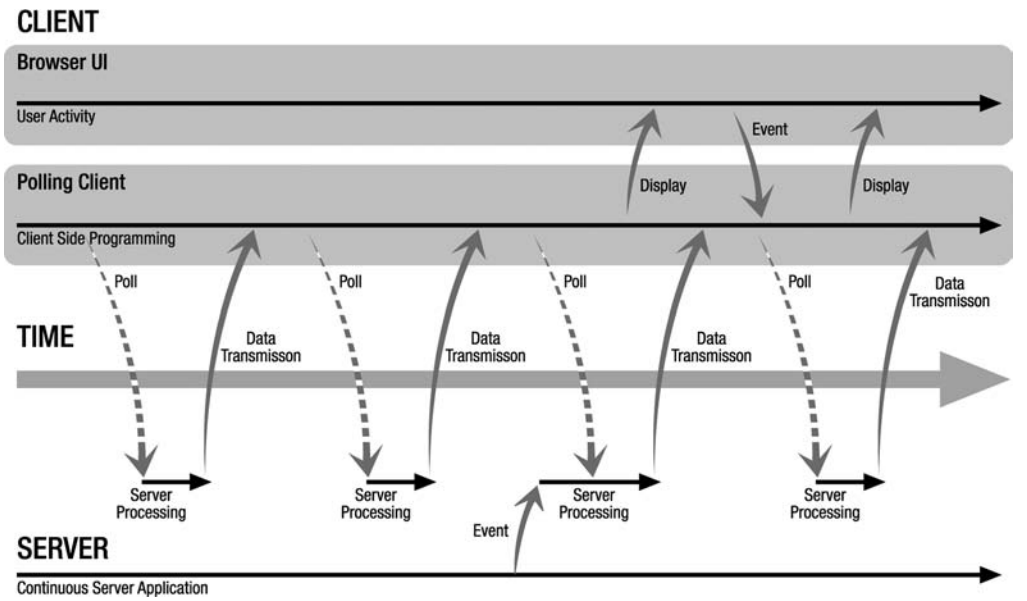


Figure 3-9. *The sequence of events in the polling technique*

Comet

Comet, the second active reverse Ajax method, is of course so named so as to be a counterpart cleaning agent to Ajax (give that a second to process if it didn't make sense, and if you still don't get it, take a break, head on down to your local supermarket, and look through the products in the cleaning supplies aisle . . . although it may not be something found in every country, as my technical reviewer pointed out, so I'll save you the time and tell you that Comet is a cleaning product sold in the United States that comes in a distinctive polished green can), and it is adequately, if a little confusingly, described in Figure 3-10.

The basic concept is very simple: a client makes a request to the server, and the server starts to respond. It then continues to respond at an exceedingly slow pace. So slow in fact that the connection is held open indefinitely because, while it's moving very slowly, it's still active. Think about that: we've essentially gotten around the HTTP limitation of not having a static connection from client to server! Data can then flow across this connection in both directions, giving you the potential for truly real-time updates initiated by the server.

Many people don't consider the polling technique to be reverse Ajax at all, and that's not entirely unfair, but Comet most definitely is, and in fact was the catalyst for the term reverse Ajax becoming well known in the first place.

The term Comet appears to have first been coined by Alex Russell, the original brains behind the very popular Dojo toolkit (`http://dojotoolkit.org`). I just wanted to give credit where credit was due . . . for all DWR's cool reverse Ajax support, including Comet, it wasn't an invention of Joe or the rest of the DWR folks (although there is an argument to be made that they've done it best to date).

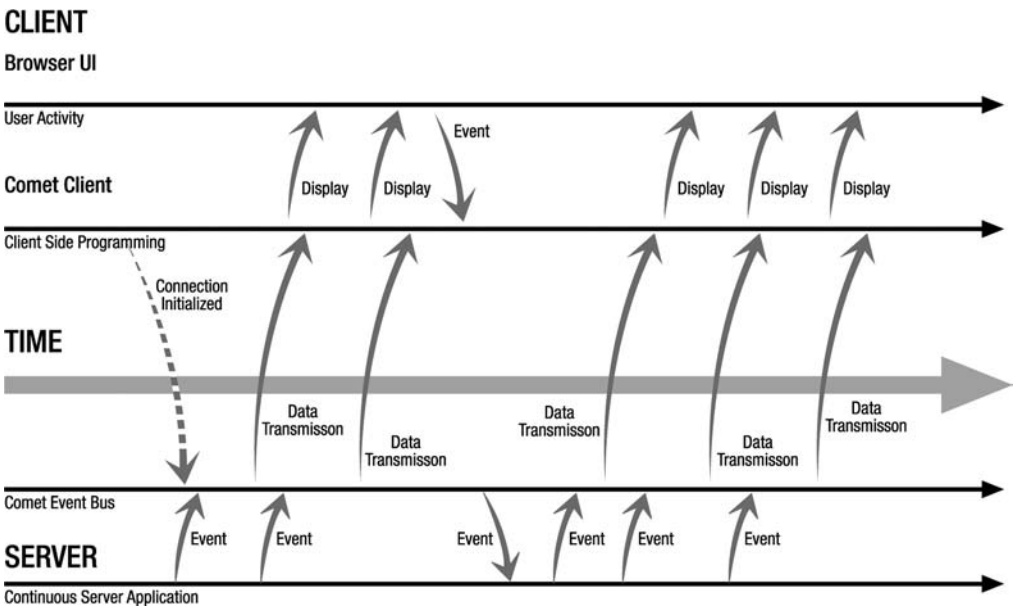


Figure 3-10. *Comet sequence of events*

If you've done any amount of enterprise development, you may well be asking a question or two right now: Does reverse Ajax, Comet specifically, scale? Will it kill my server? The answer is a resounding **maybe** on both counts.

It should be clear that holding connections open for any length of time is potentially a recipe for overloading a server and running out of request processing threads. This is referred to as *thread starving* the application server, and it is most definitely a Bad Thing™. Fortunately for us, DWR has mechanisms to deal with this.

DWR offers three different modes of reverse Ajax (aside from the polling/Comet/piggybacking modes, two of which we've seen and one—piggybacking—that we'll see next . . . call these three new modes "submodes" if you wish) when you're dealing with the active schemes. The first mode is *full-streaming* mode. This is the default mode when reverse Ajax is turned on, and it works by closing the connections every 60 seconds or so to ensure the browser is still active (at which point the connection would be automatically re-created to give the appearance of an always-active connection). DWR also implements a rudimentary form of throttling on the server where it will dynamically adjust the connection time and timeout based on the server load.

The second mode is *early closing* mode. In this mode, things work similarly to full-streaming mode except that the connection is held open only for 60 seconds if there is no output to the browser. Once output occurs, DWR pauses for some configurable time before closing the connection, forcing proxies to pass any messages on. You can configure the time delay between the first output from the server and the close of the connection, which by default is 60 seconds as in full-streaming mode.

The downside of early closing mode is that for applications with a high rate of output, it can cause a huge hit count, and even in applications with a low output rate, an event could cause all connected browsers to simultaneously reconnect. In situations with a low output

rate, but a large number of browsers, the server load can be reduced by switching to polling mode.

If the configured early closing mode time from first output to connection close is the default of `-1`, or any value greater than `1000` (1 second), it should be noted that DWR will use an iFrame on Internet Explorer to allow for streaming back from the server. This will result in the familiar page reload "click" sound being heard continually, so if you are going to use the early closing mode, you will probably want to configure the delay to some value other than the default and lower than `1000`, which avoids the clicking.

The third is actually polling, which as it turns out kind of falls into a number of categories simultaneously!

Piggybacking

The last approach to reverse Ajax, as shown in Figure 3-11, is the piggyback approach. This approach is the one "passive" method offered by DWR. It is called passive because there is a user interaction that is required to make it work (it actually could work without user interaction in theory, but it typically doesn't). In fact, for all intents and purposes, piggybacking and passive reverse Ajax are pretty much synonymous since there really isn't any other form of passive reverse Ajax that I'm aware of.

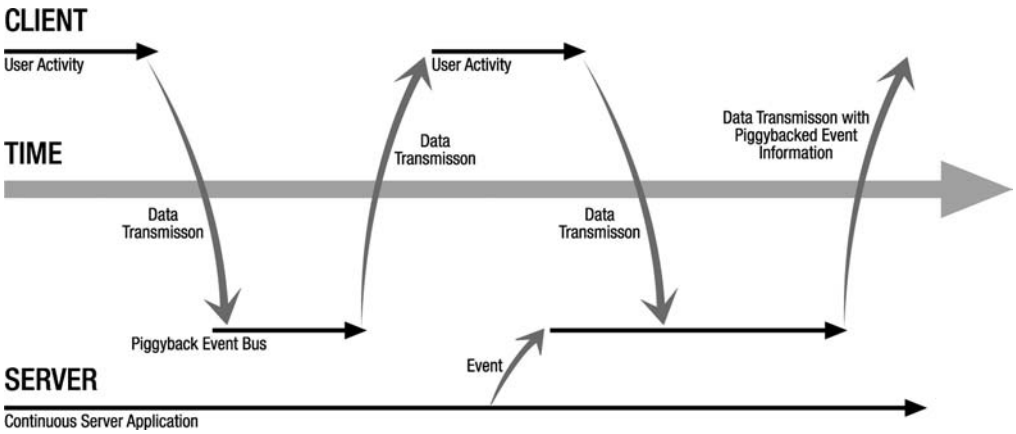


Figure 3-11. *The piggybacking technique sequence of events*

In this model, the user performs some action that results in an Ajax request being made. The server produces the appropriate response, but *in addition*, it adds something else not strictly related to the operation being performed. It's sort of akin to having a conversation with your barber who says, "OK, I'm going to wash your hair now, and another thing, it's time to get the oil changed in your car." That may be an odd conversation with your barber, but in the world of piggyback reverse Ajax, it would be par for the course!

The benefit to piggybacking as a passive technique is that no additional load is placed on the server. The server is still awaiting a request from the client, but the difference is that the server will have been queuing responses all along, and will send them as appropriate with the next request from the client. It's a very opportunistic way of doing reverse Ajax, but it doesn't have the potential to overload your server (unless so many events are being queued up that you run out of memory, but that's a whole other concern).

DWR will use piggybacking by default if you don't tell it to otherwise, so by default you don't run any risk of overloading your server by turning reverse Ajax on.

Speaking of which, how exactly do we turn reverse Ajax on, and configure the various options I mentioned and make use of it in code? As I've said on numerous occasions throughout this text, DWR makes it very simple indeed; let's see how now.

The Code of Reverse Ajax

The first step to implementing reverse Ajax with DWR is some new configuration elements (you can see all of this in action in the `reverseajax` webapp included in the source download bundle). First, in `web.xml`, a few new init parameters to the DWR servlet are required:

```
<init-param>
  <param-name>activeReverseAjaxEnabled</param-name>
  <param-value>true</param-value>
</init-param>
<init-param>
  <param-name>org.directwebremoting.extend.ServerLoadMonitor</param-name>
  <param-value>org.directwebremoting.impl.PollingServerLoadMonitor</param-value>
</init-param>
<init-param>
  <!-- This should be disconnectedTime, but because of a bug prior to the -->
  <!-- as yet unreleased 2.0.2, you have to use timeToNextPoll.  Use -->
  <!-- disconnectedTime from 2.0.2 onward. -->
  <param-name>timeToNextPoll</param-name>
  <param-value>1000</param-value>
</init-param>
```

In fact, only the first, `activeReverseAjaxEnabled`, is actually required. The other two are optional (the second, the `ServerLoadMonitor` one, is required if you're using the polling technique). Also, take note of the comments related to the `timeToNextPoll` parameter. You are actually supposed to use the parameter named `disconnectedTime`, but because of a bug in versions of DWR prior to 2.0.2 (which has not been released at the time of this writing), you have to use `timeToNextPoll` instead. Keep this in mind if you upgrade DWR versions, lest you have busted applications and a headache from trying to figure out why.

With this configuration out of the way, one new bit of JavaScript is required on the page that will be involved with the reverse Ajax, and it is simply this: `dwr.engine.setActiveReverseAjax(true);`. This, plus the configuration in `web.xml`, is all that's required to activate reverse Ajax.

If you were to add this to an existing DWR webapp and fire it up, then access it with Firefox and look at the Firebug console, you would notice that every second, a new request is going across the wire. See, it really **is** polling!

The next part of the equation is actually doing something with those polling requests. The way you do this is by writing some code on the server that updates the session for each client attached to the server. DWR keeps track of each client that contacts it by storing a session for each. This is different from the usual HTTP session (although I wouldn't be surprised if the HTTP session is in fact used internally by DWR). With it you can call JavaScript code, and the next poll request that comes in will be notified of those calls. Let's look at that code now:

```
String currentPage = wContext.getCurrentPage();
Collection sessions = wContext.getScriptSessionsByPage(currentPage);
Util utilAll = new Util(sessions);
utilAll.setValue("divTest", d.toString(), true);
```

Yes, that's really it! Once you have the name of the current page (as known to DWR), you can get a list of all the sessions currently connected to that page. You can then get an instance of the `Util` class, which is the main interaction point in DWR between your Java code and the JavaScript on the client. Passing it the list of sessions allows it to interact with them all without you having to do anything else like iterate over the collection. The `Util` class exposes a number of convenient methods, one of which is `setValue()`. This is akin to doing `document.getElementById("divTest").innerHTML = "";` on the client, but it will take care of the details such as if the target element is a text box or something else. Here, we tell it to update the contents of `divTest` using the current value of the `Date` field we record, and we also specify that we want any HTML escaped by passing `true` as the third parameter so we don't break anything client side (not really a risk here, but better safe than sorry).

In the context of the polling webapp, this code runs within a thread that updates the time (and the connected clients) once every second. Spawning threads in a J2EE container is a generally frowned-upon practice, but for an example like this it works just fine.

DWR has a chat example application that gives another example of this. It is perhaps the smallest, in terms of lines of code, implementation of a multiuser chat application in the world today (it's gotta be close in any case).

If you don't want to use the polling approach, you can switch to Comet very easily: simply comment out the following element in `web.xml`:

```
<init-param>
  <param-name>org.directwebremoting.extend.ServerLoadMonitor</param-name>
  <param-value>org.directwebremoting.impl.PollingServerLoadMonitor</param-value>
</init-param>
```

Since Comet is the default, that's all you need to do to activate it. If you do that and then run the app and again bring it up in Firefox and look in the Firebug console, you'll see exactly two requests. The first is when the page loads (kicking off the thread) and the second, which you'll note never completes, is the reverse Ajax thread (Comet). The little spinner, which indicates activity in the Firebug console, will keep spinning forever. You now have a constant connection to the server!

Finally, to see the piggyback technique in action, you have only to remove the following elements from `web.xml`:

```
<init-param>
  <param-name>activeReverseAjaxEnabled</param-name>
  <param-value>true</param-value>
</init-param>
<init-param>
  <param-name>org.directwebremoting.extend.ServerLoadMonitor</param-name>
  <param-value>org.directwebremoting.impl.PollingServerLoadMonitor</param-value>
</init-param>
<init-param>
  <!-- This should be disconnectedTime, but because of a bug prior to the -->
  <!-- as yet unreleased 2.0.2, you have to use timeToNextPoll.  Use -->
  <!-- disconnectedTime from 2.0.2 onward. -->
  <param-name>timeToNextPoll</param-name>
  <param-value>1000</param-value>
</init-param>
```

Then, somewhere in `index.jsp`, add the following:

```
<input type="button" onClick="RemoteClass.startPolling();">
```

Now if you load up the page, you'll notice that the time doesn't automatically change. However, each time you click the button, you'll see that it does change. This is in spite of the fact that the code that does the updating **is not** executed when you call the `startPolling()` method. What's happening is that the thread continues to run and queues up changes to the time. Then, when the next DWR request comes in as a result of the button press, DWR pounces on its opportunity to send the necessary updates that have been building up in the queue. This is obviously the least responsive way to do reverse Ajax, but it's also the least worrisome from a server load perspective, and so might be desirable when truly real-time updates are not actually required (this is often called "good enough" time).

As you can see, DWR offers a number of options to support reverse Ajax, all of them being very easy to implement with a lot of configuration flexibility. We'll be seeing more of reverse Ajax in action in the projects to come, but you're now armed with enough knowledge to put it to effective use, so let's head on to the next topic, which is integration with other frameworks.

Don't Go It Alone: Integration with Frameworks and Libraries

DWR has the ability out of the box to integrate with a number of well-known libraries and frameworks, to one degree or another. In most cases this boils down to being able to create objects of a given type supplied by another framework or toolkit and call methods on it. Let's have a look at each supported product and see what DWR offers us.

Spring

As you may well know already, the Spring Framework (`http://springframework.org`) is a very popular library that covers quite a bit of ground, but it started out (and became well known for) one component in particular: its IoC container. If you've ever heard the term *Spring beans*, this is where it comes from: any Java object (bean) that Spring creates automatically and injects into your classes is called a Spring bean.

Spring has actually expanded quite a bit since its early form and does a great deal more now than just IoC, but for the sake of this discussion that's all we're really interested in. However, it's certainly worth some of your time, in my opinion, to head on over to the Spring web site and check out all it has to offer. In a nutshell, Spring is concerned with allowing you to develop applications using POJOs (Plain Old Java Objects), as opposed to something like EJBs. In the process, it hides much of the complexity of doing this from developers, letting them focus squarely on the problems at hand.

Where DWR and Spring meet is in the ability of DWR to create beans via Spring and then call methods on it. DWR provides the `spring` creator, which knows how to look up a given bean in the `beans.xml` file, `applicationContext.xml`, depending on which version of Spring you use (which is where you configure your Spring beans), and instantiate it.

All you need to do is create an entry such as the following in `dwr.xml`:

```
<create creator="spring" javascript="MyBean">
  <param name="beanName" value="MyBean" />
</create>
```

The `beanName` parameter specifies the name of the bean as configured in `beans.xml`. Everything else works in the same way as you are already familiar with.

There is one other part to this equation, and that's finding your Spring configuration. There are a number of ways to do this, but perhaps the easiest is to add a listener to your webapp configuration like so:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/classes/beans.xml</param-value>
</context-param>
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

When the application starts up, this listener will execute and will configure Spring using the configuration file pointed to by the context parameter. Once that's done, you're all set!

An alternative approach is to use an extra `location` parameter in the `dwr.xml` file for each bean you want to create. This parameter points to your `beans.xml` file. This is less desirable though because it means more to change should you ever move the file, but it does serve the same purpose. Lastly, there is a programmatic way to get the same effect, by using the `SpringCreator`'s `setOverrideBeanFactory()` method. Any code in this book that uses Spring will be using the listener though, so if the other two approaches appeal to you, I suggest perusing the DWR documentation to get the specifics.

JSF

JSF (java.sun.com/javaee/javaserverfaces) is the one endorsed (because it's part of the JEE spec) framework for web application development in the Java world (don't take that to mean it's necessarily the best choice, just that it's the only "official" framework in the Java world), and DWR would be remiss to not support it. Fortunately, it does!

DWR's support of JSF is very similar to that of Spring in that it allows you to remote your managed JSF beans, and not much more. This can certainly be enough though! In classic DWR form, using this capability is very easy:

```
<create creator="jsf" javascript="ScriptName">
  <param name="managedBeanName" value="beanName"/>
  <param name="class" value="your.class"/>
</create>
```

You'll also need a new filter added to web.xml to activate the JSF support and be able to use the JSFCreator, and that entry is this:

```
<filter>
  <filter-name>DwrFacesFilter</filter-name>
  <filter-class>uk.ltd.getahead.dwr.servlet.FacesExtensionFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>DwrFacesFilter</filter-name>
  <url-pattern>/dwr/*</url-pattern>
</filter-mapping>
```

There's not much more to say about it than this. Once you have the filter in place and the appropriate entries in dwr.xml as shown here, you'll have access to your managed beans from your FacesContext, same as any other classes.

WebWork/Struts 2

DWR also offers integration with the WebWork framework from OpenSymphony (www.opensymphony.com/webwork). This support allows you to remote your WebWork Actions just like any other class.

Note that WebWork has now become Struts 2, and this integration should essentially work the same (although mention of xwork.xml refers to struts.xml now, and there could be some other minor differences).

Using this feature of DWR involves a couple of steps, but none significantly different from what we've seen already. First, you'll need some new entries in dwr.xml, as follows:

```
<create creator="none" javascript="DWRAction">
  <param name="class" value="org.directwebremoting.webwork.DWRAction"/>
  <include method="execute"/>
</create>
<convert converter="bean" match="org.directwebremoting.webwork.ActionDefinition">
  <param name="include" value="namespace,action,method,executeResult" />
</convert>
<convert converter="bean" match="org.directwebremoting.webwork.AjaxResult"/>
```

This makes DWR aware of and capable of dealing with the various types of objects that are involved. Also, if your Action invocation returns Action instances rather than the more typical pure text, you'll need one additional bit of configuration in `dwr.xml`:

```
<convert converter="bean" match="<your_action_package>.*"/>
```

Include as the value of the `match` attribute the package containing the Action instance that will be returned.

Once the configuration is in place, you'll need to import the usual DWR JavaScript into the JSP that will be calling on the Actions. In addition, you'll need to import `DWRActionUtil.js`, which is some helper code needed to work with WebWork Actions.

Invoking an Action is very much like invoking any other remote class, except that you do so through the `DWRActionUtil` object, as shown here:

```
DWRActionUtil.execute(id, params, callback [, displayMessage]);
```

The parameters to this call are as follows:

- `id`: This is one of two things. It can be the Action URI, sans the usual `.action` extension. Or, it can be an `actionDefinitionObject` JavaScript object. In this case, this object must specify the following fields: `namespace`, which is the namespace of the Action in `xwork.xml`; `action`, which is the name of the Action in `xwork.xml`; and `executeResult`, which is either `true` or `false`, indicating whether to execute the returned Action instance if the return from the method invocation is an Action instance or just return it.
- `param`: This is one of three things. If you have no parameters to pass, it is simply an empty object, `{ }`. Or, it can be the ID of a single field whose value will be transmitted to the Action invocation. Or, it can be the ID of a form, in which case all values from the form will be transmitted. Note: if your configuration in `xwork.xml` uses the `ParameterInterceptor`, then your Action will be fully initialized with the incoming values, the same as if it were being called through WebWork itself.
- `callback`: This is the usual callback function as seen elsewhere in DWR.

Struts "Classic"

Struts "Classic" (`http://struts.apache.org`), that is, Struts prior to the WebWork merger, is also supported by DWR. However, I'm not entirely sure how useful this integration is to be honest, but I'm just here to tell you about it; **you** can decide if it's worth anything to you!

DWR allows you to create and call methods on your Struts form beans, and all it takes is a new creator:

```
<allow>
  <create creator="struts" javascript="ScriptName">
    <param name="formBean" value="formBeanName"/>
  </create>
</allow>
```

If it seems a little weird that form beans are instantiated and not Actions, you aren't alone in thinking that. The problem with calling Actions in the Struts 1 (or "Classic," as some would

call it) world is that the `execute()` method, or any method used in a `DispatchAction`, requires a form bean, an Action mapping, a request, and a response object. So how would those get there? Clearly, DWR can handle the last two, but the first two it can't. That's why the Struts creator doesn't work with Actions, even though that would seem to make the most sense. That makes this creator's usefulness a little dubious in my opinion.

If you want to use Actions, you have three choices. One is to create your own creator that knows how to feed the `execute()` method the objects it needs. Another is to have the `execute()` method call on other methods in the Action that don't require those objects, so you in effect abstract out the bits that aren't Struts-specific to methods you could remote with DWR (because remember, a Struts Action is just a class that you could remote like any other). The third, and probably the best option, is to refactor your Actions so that the functionality you'd want to remote with DWR is in some delegate classes that the Action itself calls in (which is usually how you want to write a Struts application in the first place).

I should also point out that there is active work with regard to better integration with Struts, although that work seems to be focused on Struts 2, which is the codebase resulting from the merge with WebWork.

One consideration is that when you do this, assuming Struts is running alongside DWR (which is a perfectly reasonable thing to do mind you, it's just two different servlets running in the same webapp), you need to be sure that Struts starts up before DWR does. So, in `web.xml`, you'll need to specify that the `<load-on-startup>` init parameter for the Struts `ActionServlet` is a lower than that for the `DWRServlet`.

Note that official support for Struts 2 in DWR has not yet been announced, but that support will almost certainly look much more like the integration with WebWork than with this Struts "Classic" integration. In fact, a post a short while ago on the DWR mailing list indicated that the original author of the WebWork integration code in DWR hasn't yet had time to explore Struts 2 integration, and if anyone wanted to jump in and see what changes might be needed, that help would be greatly appreciated. So, if you're by chance feeling the open source itch and want to get involved, sign up for the DWR mailing list and start hacking some code! You'll love getting your name in the contribution list for the first time, that's for sure!

Beehive

Beehive (`beehive.apache.org`) is yet another web application framework from the Apache Foundation (do those cats do anything **but** web application frameworks these days?), and DWR integrates with it as well. This integration amounts to not anything more than a new creator that knows how to create Beehive `PageFlow` objects. The creator is used like so:

```
<allow>
  <create creator="pageflow" javascript="ScriptName" />
</allow>
```

Yes, that's truly it! No filters to configure, no extra converters to add, just use the `pageflow` creator, and you're good to go.

Hibernate

Now we come to the bookend of this section in integration: Hibernate (bookend in the sense that we began with Spring, which is ostensibly not a web application framework, although it does contain components of one, then we tackled a number of web application frameworks and the support DWR offers for them, and now we end with another non-web application framework, Hibernate).

The key to using the Hibernate support in DWR is to ensure that your Hibernate configuration works independently of DWR (which is generally outside the scope of this book, although in a later chapter we'll in fact get into it a little bit). Hibernate is a great piece of technology, but it can sometimes be a little tricky to get working how you want, so if you don't have everything working properly with it alone before you introduce DWR into the mix, you're in for a lot of headaches.

DWR introduces two new converters for using Hibernate, and they are the very <sarcasm>creatively named</sarcasm> `hibernate2` and `hibernate3` converters (in earlier versions of DWR, there was a single `hibernate` converter that worked for both Hibernate 2 and 3, but in later versions there are two, and the latter is the situation we'll be dealing with in this book). They are both implemented by the `HibernateBeanConverter` class.

At this point you've seen enough DWR configuration that you should be able to pretty well figure out what it would look like on your own. Remember, it's nothing but a couple of new converters.

Something Old, Something New: Annotations

So far, we've dealt with DWR configuration in terms of the `dwr.xml` configuration file. This is all fine and good, but it seems that the world has collectively decided that annotations are the way to go. So, is DWR behind the times? Most certainly not my dear reader! DWR allows you to replace entirely or work in tandem with its configuration file using Java 5 annotations, and in this section we'll see exactly how.

TO ANNOTATE OR NOT TO ANNOTATE, THAT IS THE QUESTION

I said that the world has seemingly decided collectively that annotations are the cat's meow, to coin a phrase, so who am I to question the wisdom of the Java developer community as a whole? Why, I'm a guy writing a book, that's who!

Some people feel that annotations have their place for sure in the toolbox of the Java developer, but that they aren't the be-all and end-all of things (this is my opinion, but it is shared by many others, as a few minutes with Google will convince you).

The key question to ask about annotations is what they really are. Sun has always claimed they were metadata, an extension to the existing metadata in the Java APIs. Is that really what they are though? Don't they in fact decorate an object's blueprint? They tell Java something extra about your code, how to compile it, things like that. It's not data about data, which is the academic definition of the term metadata.

Let's forget that though, because it's largely a philosophical debate. The real question, and the point at which I have some reservations about annotations, is that they represent hard-coded configuration information. For how many years have we tried to abstract things like configuration out of our code? That was the

whole point of externalized configuration files in the first place. Remember that when an annotation changes, you have to recompile and in most cases redeploy the application. Isn't this the exact polar opposite of what we've been trying to do all these years? When did we decide that was the wrong direction?

Let me be clear, I'm not saying annotations are evil and should never be used. What I **am** saying, however, is that this sudden drive toward doing **everything** with annotations to me is misguided. View annotations like you would a screwdriver: sure, you can use it as a chisel, but it's not the right tool for the job (most likely). Don't just jump on the annotation bandwagon because everyone seems to be doing it . . . unless you're prepared to follow them in their jump off the Brooklyn Bridge too (and no, I'm not likening annotations to jumping to one's death, it's not **that** bad, I'm just making the comparison in terms of simply doing what everyone else is doing for the sake of doing what everyone else is doing!).

The first step to using annotations with DWR is to use a new DWR controller servlet, specified in `web.xml` as follows:

```
<servlet>
  <description>DWR controller servlet</description>
  <servlet-name>DWR controller servlet</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
  <init-param>
    <param-name>classes</param-name>
    <param-value>
      com.example.RemoteFunctions,
      com.example.RemoteBean
    </param-value>
  </init-param>
</servlet>
```

The new `classes` init parameter is a comma-separated list of all classes you intend to remote that contain annotations. Note that if you need to specify an inner class, you need to use the `$` notation rather than dot notation.

The actual annotations couldn't be simpler. Let's start with a simple class that we wish to remote, as shown in Listing 3-4.

Listing 3-4. *A Simple Class That We'll Configure with Annotations*

```
public class RemoteClass {

  public String myMethod(String inParam) {
    return inParam + " returned";
  }

}
```

Now, to annotate this and prepare it for DWR use, we have only to add a few annotations, the result being the code in Listing 3-5.

Listing 3-5. *That Same Simple Class All Annotated for DWR's Gratification*

```
@RemoteProxy
public class RemoteClass {

  @RemoteMethod
  public String myMethod(String inParam) {
    return inParam + " returned";
  }

}
```

And just like that, we can now call the `myMethod()` method of the `RemoteClass` class using DWR! The `@RemoteProxy` annotation indicates the class can be remoted, and the `@RemoteMethod` annotation indicates the method can be called.

Any method in the class not annotated will **not** be available to DWR. If you wish to have a different JavaScript object name than the name of the class (`RemoteClass` in this case), you add a `name` attribute to the `@RemoteProxy` annotation like so:

```
@RemoteProxy(name="SomeClassName")
```

When you need to deal with beans that you want to allow DWR to convert, there are the `@DataTransferObject` and `@RemoteProperty` annotations, their usage shown in Listing 3-6.

Listing 3-6. *A Simple Bean Annotated for DWR's Use As a Convertible Type*

```
@DataTransferObject
public class MyBean {

  @RemoteProperty
  private String firstName;

  public String getFirstName() {
    return firstName;
  }

}
```

The `@DataTransferObject` annotation indicates the bean can be marshaled by DWR, and the `@RemoteProperty` annotation indicates that the property `firstName` should be marshaled. So, if you wish to have some members of the bean that are not transmitted back and forth between client and server, simply leave off the corresponding `@RemoteProperty` annotation, and you're all set.

Note that there is an optional `converter` attribute to the `@DataTransferObject` annotation that can specify the type of converter to use to marshal the class. Documentation was a little sparse on this attribute frankly, so I hesitate to show an example, lest it be wrong. I can tell you that the default converter used is the ubiquitous `BeanConverter`, so in most cases you won't need this anyway, but if you do, I suggest pinging the DWR mailing list to get a description of what the attribute does. We won't need it in any of the projects in this book, so I think I can cop out a little bit here!

Incidentally, this is one of those times where using annotations doesn't hurt my head very much. I think specifying that an object is or isn't remotable, that a method is or isn't remotable, is a good use of annotations. Please do form your own opinion though! For what it's worth, you'll be seeing both the configuration file approach and the annotations approach in the projects in this book, so you'll have a good basis to form that opinion on anyway.

Summary

In this chapter, we took a look at some topics in the DWR world that are somewhat more "advanced" in the sense that you won't always need these things to write a DWR-based application (although perhaps more often than not you will). We examined how you can introduce robust security into a DWR application, and how to handle all sorts of error/exception situations. We discussed how we can access other parts of an existing web application in the course of our DWR processing. We also discussed reverse Ajax, the newer sibling to big brother Ajax. We then saw how we can integrate DWR with other frameworks to gain all sorts of added value, and finally, how we can use Java 5 annotation support in DWR to configure our applications in a more streamlined way.

And with the close of this chapter, we now move on to the second part of the book, which is comprised of the applications. Strap yourself in; it's bound to be a wild ride!

PART 2

The Projects

All I need to know about life I learned by reading poster sigs on Slashdot.

—Frank W. Zammetti

Always program as if the person who will be maintaining your program is a violent psychopath that knows where you live.

—Martin Golding

Any sufficiently advanced bug is indistinguishable from a feature.

—Bruce Brown

Once you're done writing the code, never open it again unless you want to see how incomprehensible and utterly ridiculous it really is.

—Raphael Sazonov

Documentation is like sex: when it is good, it is very, very good; and when it is bad, it is better than nothing.

—Dick Brandon

To err is human—and to blame it on a computer is even more so.

—Robert Orben

The secret of creativity is knowing how to hide your sources.

—Albert Einstein

They have computers, and they may have other weapons of mass destruction.

—Janet Reno

All of the books in the world contain no more information than is broadcast as video in a single large American city in a single year. Not all bits have equal value.

—Carl Sagan

CHAPTER 4

InstaMail: An Ajax-Based Webmail Client

This chapter will introduce an application called InstaMail, which is an Ajax-based webmail application. DWR will form the foundation of the server side of things as usual, and we'll also use some client-side DOM scripting and CSS to make this application a little more "pretty" than it otherwise might be. In the end, we'll have a web application that you can run on a server and that will give you remote access to your POP3 account from anywhere. Although it will not be as full featured as Microsoft Outlook, it will be more than capable of performing the basics.

Application Requirements and Goals

A while back, access to e-mail accounts was always through some bulky fat-client application. AOL was and still is perhaps the most famous; you may recall others like CompuServe and Delphi (true, these services have always been far more than just e-mail, however, we're discussing e-mail after all, and that's one part of the services they provided, so allow me the minor inaccuracy, won't you?). Later on, and still to this day, we have applications like Microsoft Outlook and Outlook Express, Thunderbird, and Eudora, just to name a few.

More recently though, over the past three to four years perhaps, people have started to realize that it would be nice to have e-mail available to them anywhere, not just on a PC or laptop with the client installed. That realization, along with the advent of the Internet and always-on access for many people, heralded the birth of the webmail client.

A webmail client is nothing more than a web application that runs on a server and acts essentially as a proxy between you and your POP3/IMAP account. POP3 is the most popular -mail protocol out there for mail retrieval, and Simple Mail Transfer Protocol (SMTP) is the most common for mail transmission. Webmail usually contains some basic functions, and increasingly even more advanced functions. Clearly it has to be able to send and receive messages from a POP3/SMTP account, and usually it will have some form of address book as well. Those are pretty much the basics that everyone expects to find. Things like multiple folder support, grouping of e-mail "conversations," rich e-mail editing, file attachments, and alerts are some of the more advanced features that can be found in many webmail applications today.

Here, we'll build ourselves a webmail client that covers those basics. Because the advanced features generally require knowing a little more about the server the application runs on than I can safely assume in the context of this book, we won't delve into many, if any, of those advanced features. However, as I am sure you can guess, they make for a very nice, easy list of suggested enhancements for you at the end!

Let's now codify the features this webmail application will support, and the general design goals:

- We want the user interface (UI) to be somewhat "pretty." Not just a boring, simplistic HTML page, but something a little easier on the eyes.
- We should, of course, be able to view our POP3 Inbox and read messages in it, as well as delete messages.
- Sending a message as well is an obvious goal. We should be able to compose a new e-mail, either to a contact in our address book or to someone else, and we should also be able to reply to a received message, complete with quoting of the original message.
- To make things simple and stay focused on what we should be focused on, InstaMail will support a single e-mail client for a single user, and there will be no security per se, i.e., no username and password will be required to access InstaMail (they will still, of course, be required, usually to access the POP3/SMTP servers).
- We'll provide an address book from which a new message can be started. We'll store just some basic information for each contact, namely first name, last name, e-mail address, and comment. Comment can be any note you want to make about the contact.
- In addition to access to the Inbox, we'll maintain all sent messages locally.
- We'll be able to delete the message currently being read, or select a batch to delete from the Inbox view.
- In addition to the UI being a bit "pretty," we want to exercise some DHTML (Dynamic HTML) and CSS skills and incorporate some slightly "fancy" features, things like some button mouse-over effects and a Please Wait float-over. We won't go completely crazy, but just enough to have fun!

That list should give us more than enough to do. It should also be a bit of a fun project to dissect, so without delay, let's get to it!

Dissecting InstaMail

Let's begin by getting a feel for the overall structure of InstaMail by looking at the file layout, shown in Figure 4-1.

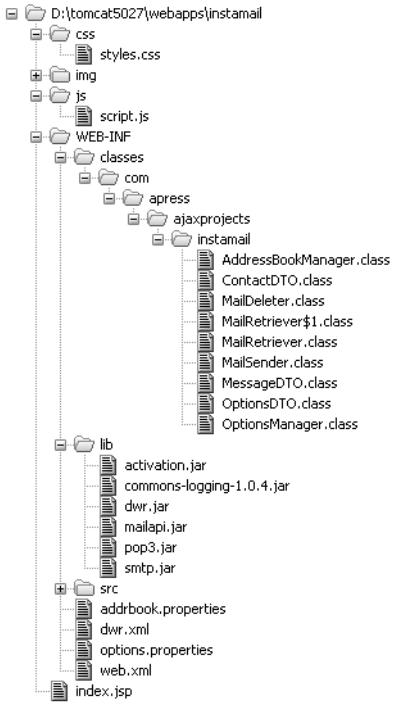


Figure 4-1. *Directory structure of InstaMail*

InstaMail consists of a single JSP page, `index.jsp`. All of the markup for the entire application is in that one file. This file imports the style sheet file `styles.css` in the `/css` directory. It also makes use of all the images in the `/img` directory, which I have not expanded here in the interest of keeping this layout image small (there are a rather large number of images involved, and listing them all would have more than doubled the height of this figure). All of the JavaScript for InstaMail, save a single `init()` function in `index.jsp`, is contained in the `script.js` file in the `/js` directory, which is also imported by `index.jsp`. We can also see here nine classes that make up the server side of the application in the `WEB-INF/classes` directory, in a typical package directory structure. Lastly, we can see the libraries that InstaMail makes use of in the `WEB-INF/lib` directory, and they are described in Table 4-1.

Table 4-1. *The JARs InstaMail Depends on, Found in* `WEB-INF/lib`

JAR	Description
`activation.jar`	JavaBeans Activation Framework, needed by the JavaMail API.
`commons-logging.jar`	Jakarta Commons Logging is an abstraction layer that sits on top of a true logging implementation (like Log4J), which allows you to switch the underlying logging implementation without affecting your application code. It also provides a simple logger that outputs to `System.out`, which is what this application uses.
`dwr.jar`	The JAR containing all the DWR classes.

Continued

Table 4-1. *Continued*

JAR	Description
`mailapi.jar`	The JavaMail API, the Sun-standard library for performing mail functions.
`pop3.jar`	JavaMail extensions for dealing with POP3 mail servers.
`smtp.jar`	JavaMail extensions for dealing with SMTP mail servers.

The `WEB-INF/src` directory contains all the source code for this project, including the Ant build script.

Configuration Files

The first thing we'll look at with InstaMail is the configuration files that tell the servlet container, and DWR, how the application works.

web.xml

First up is `web.xml`, shown in Listing 4-1.

Listing 4-1. *InstaMail's* `web.xml` *Config File*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app id="instamail" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>InstaMail</display-name>

  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
   </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

  <!-- Session timeout config. -->
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
```

```
  <!-- Welcome file config. -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Well, there certainly is not much to it, is there? The most important item is the DWR servlet definition. Here we are mapping it to the path `/dwr/*` and telling it we want to see debug information. Beyond that we are simply setting a session timeout of 30 minutes and setting `index.jsp` as our welcome page. Very concise indeed!

dwr.xml

Next up is `dwr.xml`, which is the DWR configuration file that was described in previous chapters, shown in Listing 4-2.

Listing 4-2. `dwr.xml` *for InstaMail*

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>
    <convert converter="bean"
      match="com.apress.dwrprojects.instamail.MessageDTO" />
    <convert converter="bean"
      match="com.apress.dwrprojects.instamail.OptionsDTO" />
    <convert converter="bean"
      match="com.apress.dwrprojects.instamail.ContactDTO" />
    <create creator="new" javascript="OptionsManager">
      <param name="class"
        value="com.apress.dwrprojects.instamail.OptionsManager" />
    </create>
    <create creator="new" javascript="MailRetriever">
      <param name="class"
        value="com.apress.dwrprojects.instamail.MailRetriever" />
    </create>
    <create creator="new" javascript="MailSender">
      <param name="class"
        value="com.apress.dwrprojects.instamail.MailSender" />
    </create>
    <create creator="new" javascript="MailDeleter">
      <param name="class"
        value="com.apress.dwrprojects.instamail.MailDeleter" />
    </create>
```

```
    <create creator="new" javascript="AddressBookManager">
      <param name="class"
        value="com.apress.dwrprojects.instamail.AddressBookManager" />
    </create>
  </allow>
</dwr>
```

Like `web.xml`, this is not exactly a *War and Peace*–sized piece of code! To remind you, `dwr.xml` basically tells DWR, among other things, what classes it is allowed to create and access, and also what objects it can convert to and from JavaScript and Java. In this case we are listing the Data Transfer Objects (DTOs) that InstaMail uses as convertible, and we are listing the five main classes that contain the server-side functionality of the application. We'll of course be looking at them in detail soon, but their names alone should give you a very clear idea of what they do. Note that as shown here, there is no limitation on what methods or properties of the classes can be accessed. The only limitation is on those classes not listed, which cannot be remotely called at all.

The Client-Side Code

Before we dive into the code that makes up the client side of InstaMail, let's take our first brief look at the application itself, as shown in Figure 4-2. This is the first screen you should see upon running InstaMail for the very first time.

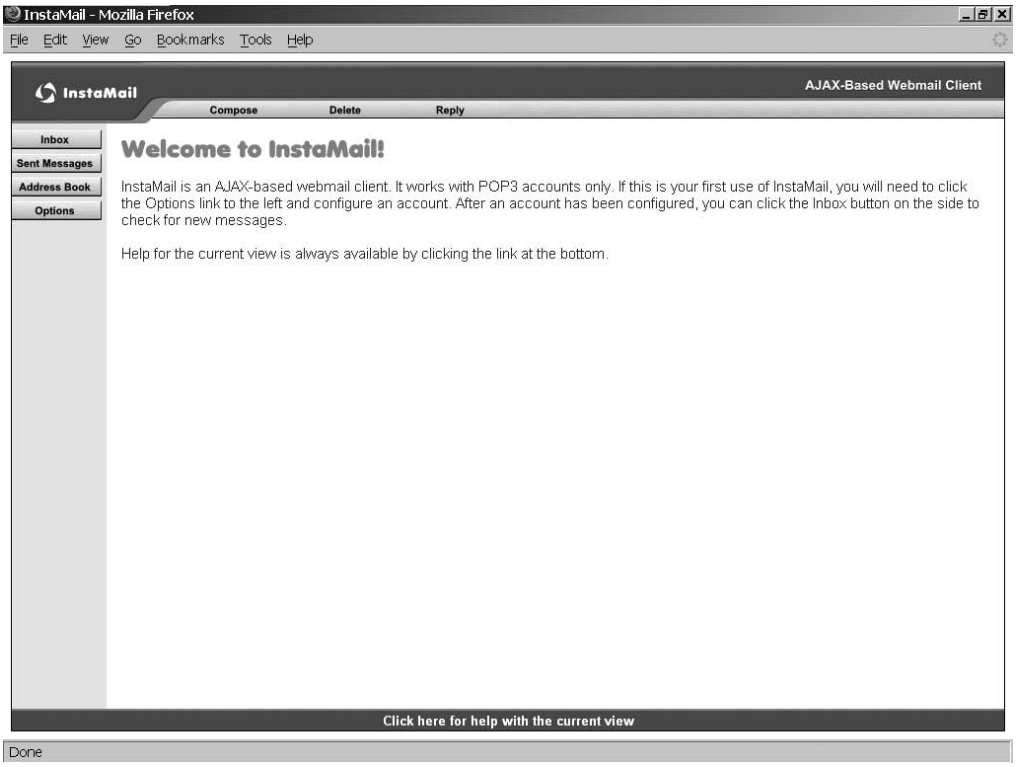

InstaMail - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
InstaMail
AJAX-Based Webmail Client
Compose
Delete
Reply
Inbox
Sent Messages
Address Book
Options
Welcome to InstaMail!
InstaMail is an AJAX-based webmail client. It works with POP3 accounts only. If this is your first use of InstaMail, you will need to click the Options link to the left and configure an account. After an account has been configured, you can click the Inbox button on the side to check for new messages.
Help for the current view is always available by clicking the link at the bottom.
Click here for help with the current view
Done

Figure 4-2. *The InstaMail "intro" view*

InstaMail is a fairly good-looking application if I do say so myself! It is always worth noting that engineers often don't have the artistic eye that our web designer brethren possess. Even so, the engineers among us should always strive to do the best with the assets we have, and that is precisely what I have attempted to do here.

We should also not be ashamed to use the tools available to us! The basic page layout and graphics for InstaMail were generated with a program called Xara Webstyle, which is a god-send for those of us who are not artists at heart (I can barely write my own name clearly, or draw a good straight line without a ruler!). Taking the basics that Webstyle provides, and after tweaking and extending it, I arrived at a look and feel that I'm fairly happy with, and I hope you will be too.

As it is shown here, InstaMail has not been used before, and therefore all you can do is go to the Options view to set up your e-mail account. If you had already done that, then you would see the row of buttons along the top and on the side that you will see in later views, and the Getting Started verbiage would not be present at all. Note the link at the bottom. You can click it at any time to get help on the current view.

We'll be looking at the other screens that make up InstaMail as we go through it, but this screenshot gives you a feel for the overall look of the application (hopefully you have already played with it a bit; if not, you probably should at this point). For now though, we'll concern ourselves with the code behind it all, so let's get right to that now.

styles.css

The first bit of code we're going to look at is the style sheet that InstaMail uses. Because it's fairly long, I will not list it in its entirety here (although you should take some time to review it on the side, just for the sake of a complete picture of what makes this application tick). However, it contains some interesting bits that I will call your attention to. First, here are some selectors that are used for the hover effect of the help link at the bottom:

```
.footer {
  font-size           : 10pt;
  font-weight         : bold;
  font-family         : Arial;
  color               : #ffffff;
  background-color    : #2661dd;
  border-top-color    : #ff7f00;
  border-top-width    : 2px;
  border-top-style    : solid;
  padding-top         : 2px;
  padding-bottom      : 2px;
}


.footerHover {
  font-size           : 10pt;
  font-weight         : bold;
  font-family         : Arial;
  color               : #000000;
  cursor              : pointer;
```

```
  background-color    : #ff7f00;
  border-top-color    : #ff7f00;
  border-top-width    : 2px;
  border-top-style    : solid;
  padding-top         : 2px;
  padding-bottom      : 2px;
}
```

Notice how the border is set on the top only? That is how the effect of having a divider line above it is achieved. The hover state alters the cursor, showing a pointer (or hand, depending on browser and OS) as well as altering the background color. This is a nice, simple, active feedback response for the user. This same effect is used within the message lists:

```
.msgListRow1 {
  font-size           : 12pt;
  font-weight         : normal;
  font-family         : Arial;
  color               : #000000;
  background-color    : #ffffff;
  cursor              : default;
}

.msgListRow2 {
  font-size           : 12pt;
  font-weight         : normal;
  font-family         : Arial;
  color               : #000000;
  background-color    : #efeff7;
  cursor              : default;
}

.rowHover {
  font-size           : 12pt;
  font-weight         : normal;
  font-family         : Arial;
  cursor              : pointer;
  background-color    : #ff7f00;
}
```

Why two selectors for the nonhover state, you ask? This is because of the row striping it. The rows of the result list alternate between shaded and nonshaded. This is a nice thing to do for users because it lets them track information across the screen easily, allowing them to keep the data that goes together straight in their minds. One way to accomplish this is to alternate what selector is used on a given row. Here, the rows alternate between using `msgListRow1` and `msgListRow2`.

One last selector I'd like to draw your attention to is the one used for the Please Wait float-over. A float-over, or simply a floating layer, is a layer that floats over others—that is, has a higher `z-index`. Note that this differs from the `float` CSS attribute, which describes where one

element is placed in relation to another vertically or horizontally. In this instance, float-over refers to the layer's position *on top* of other layers—that is, it "floats" over them.

Even though when you perform an operation in InstaMail that results in the Please Wait box being shown the layer behind it is hidden, it is still hovering over the layer, and this selector accomplishes that.

```
.cssPleaseWait {
  font-size           : 16pt;
  font-weight         : bold;
  font-family         : Arial;
  color               : #ffffff;
  position            : absolute;
  left                : 1px;
  top                 : 1px;
  width               : 330px;
  height              : 66px;
  background-color    : #ff7f00;
  display             : none;
  z-index             : 1000;
  border-top-width    : 4px;
  border-right-width  : 4px;
  border-left-width   : 4px;
  border-bottom-width : 4px;
  border-right-style  : solid;
  border-bottom-style : solid;
  border-top-style    : solid;
  border-left-style   : solid;
  border-right-color  : #2161de;
  border-bottom-color : #2161de;
  border-top-color    : #2161de;
  border-left-color   : #2161de;
}
```

The part that really makes it float over everything else is the `z-index`. This CSS attribute sets the depth of a given layer. For instance, if you have three `<div>`s on a page, all absolutely positioned at the same location, which one would you see? If you visualize the structure of those layers, it is a stack, three layers on top of each other. Which one is on the top of the stack? The answer is that it will always be the one with the highest `z-index` value. In this case, we give the Please Wait layer a `z-index` of `1000` so that it will definitely be on top of everything else.

index.jsp

The next piece to look at is `index.jsp`, which is the only document in InstaMail! Once again, because of its length, I will not list the entire source here but just call out details. The first detail we should look at is found in the `<head>` of the document, and is a collection of JavaScript imports:

```
<!-- DWR interfaces. -->
<script type="text/javascript" src="js/script.js"></script>
<script type="text/javascript" src="dwr/interface/OptionsManager.js"></script>
<script type="text/javascript" src="dwr/interface/AddressBookManager.js"></script>
<script type="text/javascript" src="dwr/interface/MailRetriever.js"></script>
<script type="text/javascript" src="dwr/interface/MailSender.js"></script>
<script type="text/javascript" src="dwr/interface/MailDeleter.js"></script>
<script type="text/javascript" src="dwr/engine.js"></script>
<script type="text/javascript" src="dwr/util.js"></script>
```

Notice that the path for each of these contains the `/dwr/` portion. That means that these will be served by the DWR servlet. Pause and think about that for a moment, because it is really pretty neat . . . DWR is serving JavaScript! You see, DWR, using the information provided in `dwr.xml`, generates some JavaScript code on the fly by using reflection to examine the classes we want to remote. Notice that the name of the requested JavaScript file in each line matches the name of one of the classes in `dwr.xml`. This is quite deliberate. The code that the servlet serves, and which we import into our page, is what allows us to call on methods of an object on the server as if it were on the client. Very cool!

I also want to mention that all the DWR documentation seems to show these imports with the context in the `src` URL, `/instamail/dwr/interface/OptionsManager.js`, for instance. This is not necessary, and it is in fact better I think to leave them relative, as shown here, so that changing the context name will not cause the application to break.

Immediately following those imports is a `<script>` block that contains a single function, `init()`, as shown here:

```
<script>

  // Initialize the application.  This has to be here as opposed to
  // script.js because there is some JSP scriplet here that has to execute,
  // and that wouldn't have happened if it was in an external JS file.
  function init() {
    // There was a strange problem with Firefox... in some cases, no matter
    // what I did, the text boxes would still have values when I reloaded
    // the app. I never did figure out why this was, so all the text boxes
    // in the app are cleared here, which deals with the issue.  Same thing
    // for enabling text fields.  If anyone figures out why this happens,
    // I'd love to hear it!  Until then, I chalk it up as a Firefox quirk.
    setValue("contactNameEntry", "");
    setValue("contactAddressEntry", "");
    setValue("contactNoteEntry", "");
    setValue("pop3ServerEntry", "");
    setChecked("pop3ServerRequiresLoginEntry", null);
    setValue("pop3UsernameEntry", "");
    setValue("pop3PasswordEntry", "");
    setValue("smtpServerEntry", "");
    setChecked("smtpServerRequiresLoginEntry", null);
    setValue("smtpUsernameEntry", "");
    setValue("smtpPasswordEntry", "");
```

```
    setValue("fromAddressEntry", "");
    setValue("composeToEntry", "");
    setValue("composeSubjectEntry", "");
    setValue("composeTextEntry", "");
    setDisabled("contactNameEntry", false);
    setDisabled("contactAddressEntry", false);
    setDisabled("contactNoteEntry", false);
    setDisabled("pop3ServerEntry", false);
    setDisabled("pop3ServerRequiresLoginEntry", false);
    setDisabled("pop3UsernameEntry", false);
    setDisabled("pop3PasswordEntry", false);
    setDisabled("smtpServerEntry", false);
    setDisabled("smtpServerRequiresLoginEntry", false);
    setDisabled("smtpUsernameEntry", false);
    setDisabled("smtpPasswordEntry", false);
    setDisabled("fromAddressEntry", false);
    setDisabled("composeToEntry", false);
    setDisabled("composeSubjectEntry", false);
    setDisabled("composeTextEntry", false);
    // Start out on the Intro view.
    currentView = "divIntro";
    showView(currentView);
    <%
      OptionsDTO options = new OptionsManager().retrieveOptions(
        pageContext.getServletContext());
        // The application has not yet been configured, so we want to
        if (options.isConfigured()) {
    %>
          // The buttons start out not showing, to avoid Javascript errors
          // if you hover over them before the page fully loads, but now
          // we can show them.
          setClassName("topButtons", "divShowing");
          setClassName("sideButtons", "divShowing");
          appConfigured = true;
    <%
        } else {
    %>
          // The application has not yet been configured.  In this case,
          // the buttons ARE NOT shown, and instead we show the normally
          // hidden "Getting Started" div.
          setClassName("divGettingStarted", "divShowing");
          appConfigured = false;
    <%
        }
    %>
  }

</script>
```

As its name and comments imply, this function is called when the page loads to initialize InstaMail. In a bit we'll look at `script.js`, which contains the balance of the client-side code for InstaMail. I mention that because you may wonder why this script is in `index.jsp` as opposed to `script.js` with everything else. The reason is the scriptlet section at the end. If this was in the `.js` file, the server would not interpret the scriptlet, and thus the functions it performs would not work. It has to be in the JSP so that the container interprets it while rendering the HTML from the JSP.

The calls to `setValue()`, `setChecked()`, and `setDisabled()`, functions we'll examine shortly, are present to get around an issue I was seeing in Firefox. Even when reloading the page, for some reason, Firefox would continue to maintain any entries in text boxes. To this day I don't know why this was happening, although it didn't occur in IE. To get around the problem I simply wrote code to clear out all the text boxes and check boxes and such when the page loads. In some ways, this is better anyway, and more typical: most applications tend to have an initialization routine at the start, so this is adhering to that pattern.

The current view is set to the Intro view after the fields have been cleared. Lastly, the scriptlet section is encountered. The purpose of this section is simply to see if the app has been configured yet—that is, if an e-mail account has been set up. If it hasn't been configured yet, then all the buttons are hidden and the `divGettingStarted` `<div>` is shown, which is just some verbiage telling you that you need to set up an account, and a link to the Options view, since the user should only be able to configure options in this state. Once the application is configured, all the buttons are shown as normal, and `divGettingStarted` is hidden. Because we determine whether the app has been configured by getting an `OptionsDTO` object, which has a field `configured` that will be set to `true` or `false` accordingly, we need this to execute on the server. It actually could have been another Ajax call to the server, but I felt this was a much simpler and cleaner way to do it. The trend in recent years has been to move away from scriptlets in JSPs, and I generally agree with that trend. However, I don't take it to the extreme of suggesting it should never be done, and in a case like this I felt it was acceptable, and probably even more desirable because the alternative is something like a custom tag, which seems like overkill, or else an extra Ajax call as I mentioned. With the extra call, you have to make sure the UI is not accessible to the user for the duration of the call; otherwise, the user could be performing functions that really should be disabled, and which would be disabled as a result of the Ajax call. The KISS principle is something I adhere to, and this seems like a good example of following that principle.

GENE SIMMONS WOULD BE PROUD

The KISS principle, as opposed to the rock band (which would probably **not** be described by this principle), is an acronym for Keep It Simple Stupid (other oft-sited definitions include Keep It Simple Sherlock, Keep It Sweet & Simple, and Keep It Short & Simple). It is a common phrase used in software engineering (and in other engineering disciples in general) that basically means to not overthink a problem.

It is oftentimes a given that engineers and creative thinkers tend to come up with solutions to problems that are more exercises in their own ability to think "outside the box" than in solving the problem in a practical, simple manner. Java developers in particular, for whatever reason that my C final grade in Psychology class doesn't qualify me to determine, tend to do this frequently (especially, but not exclusively, those with any variation of the word "Enterprise" in the job titles!). This frequently leads to solutions that are brittle because they have more moving parts than is necessary, and which are also difficult to maintain later.

> The KISS principle is roughly equivalent to the famous Occam's Razor principle, which states (as uttered concisely by Jodi Foster's character Dr. Ellie Arroway in the movie *Contact*) ". . .that, all things being equal, the simplest answer is usually the right one." Albert Einstein also said it extremely well (you wouldn't expect him to say something poorly, would you?) when he said that "Everything should be made as simple as possible, but no simpler."
>
> It may not be the principle of choice for those who charge consulting fees by the hour, or who make a living writing research papers all day long, but for those of us who work to develop solid, maintainable, and extensible business software each and every day on time and under budget, it's one of the best guiding lights you can have.

After that comes the markup for InstaMail in its entirety. Notice as you use the application that there are a number of views, a.k.a. screens, that you can flip between. Notice too that as you do so, you are never retrieving the page from the server. From the start, all the various views of the application are in your browser. Each view is its own `<div>` on the page, and the appropriate `<div>` is shown at the appropriate time. There is the Intro `<div>`, the Inbox `<div>`, the Sent Messages `<div>`, the Message `<div>`, the Address Book `<div>`, the Options `<div>`, and the Compose `<div>`. Looking through the contents of `index.jsp`, you will literally see those `<div>`s. This seemingly simple structure yields a great deal of power and makes for a very well-performing application. Think about it: the server only returns data to us; all of the markup is already on the client. This is very bandwidth efficient, and yields fewer delays in switching views for the user. Something like clicking the Compose button is instantaneous. Even a small delay here, as would be present in a typical webapp where the server would be called upon to serve that page, would be perceived by the user in a negative way. This approach completely avoids that problem.

One thing I would like you to notice as you look through the markup is how few times you see the `style` attribute used on elements. Except for a few exceptions, only `class` is used. This is by design. The presentation has been abstracted out into the style sheet almost entirely (and truthfully it could have been abstracted out entirely). Doing this cleanly separates what your application looks like from its structure, as we have discussed before. Even this application, however, does not go as far as is possible in this regard, but it does go further than some of the other applications in this book do.

Aside from these few points, the markup is actually rather unremarkable. I therefore leave it to you to examine in your own time. I believe you will find that there is really very little going on. What you will see, though, is a lot of calls to JavaScript functions that exist in `script.js`, and that is therefore our next destination.

script.js

`script.js`, being another very long piece of code, won't be listed completely here either. I'll again call out the pieces that are pertinent, as the discussion warrants. But also like before, I suggest taking a moment now to get the "lay of the land," so to speak, by having a look at the entire source file yourself. When you're all done, come back 'round these parts, and we'll start tearing it apart together.

Setting the Table for Efficiency

The first thing you see in this file is a batch of image preloads, all looking like this:

```
img_send = new Image();
img_send.src = "img/send.gif";
img_send_over = new Image();
img_send_over.src = "img/send_over.gif";
```

There are of course a lot more of them (11 in total), but they all take this same basic form. An image preload is nothing but some JavaScript that loads an image from the server into the browser's memory. This is usually done for image rollovers, that is, when an image changes when you move your mouse over it. The preload stops the browser from having to go out to the server when your mouse moves over the image to get the alternate version. If it did this, the user would perceive a delay, which is unpleasant. This example shows preloading the two versions of the Send button: how the button looks when you are not hovering over it, and how it looks when you are.

A Few Globals Never Hurt Anyone

Following that are three global variables, described in Table 4-2.

Table 4-2. *Global Variables in* `script.js`

Global Variable	Description
`currentView`	This stores the name of the current view, that is, the name of the `<div>` that is currently visible.
`checkboxNumber`	Later on in the code, we'll be dynamically populating a table, and next to each item in the table will be a check box. Each check box needs to have a unique ID associated with it, and to do this we simply attach a number onto the end of its name. This variable is used during that process. With the way the table creation is done using DWR, making this variable global seems to be the only way to persist its value across function calls, which is what we need to do to make this work.
`appConfigured`	This is set during initialization, and it is used later on when a determination about the app being configured or not is required in the JavaScript.

Next up we see two functions that handle our mouse events for our buttons on the top and side:

```
// onMouseOver handler for buttons.
function btnMouseOver(obj) {
  id = obj.id;
  obj.src = eval("img_" + id + "_over").src;
  obj.style.cursor = "pointer";
}
```

```
// onMouseOut handler for buttons.
function btnMouseOut(obj) {
  id = obj.id;
  obj.src = eval("img_" + id).src;
  obj.style.cursor = "";
}
```

This is pretty typical mouse rollover code. We get the ID of the object that called the method, one of the buttons, and then use the `eval()` function to get a reference to the appropriate preloaded image. We then set the `src` of the button to the `src` of the preloaded image, completing the rollover (or rollout, as it were). In addition, the `cursor` style attribute is changed so that we get a pointer when we hover over a button.

Make the Mouse Do Pretty Things

After that are two similar functions for dealing with rows in either the Inbox or Sent Messages tables:

```
// onMouseOver handler for rows in Inbox or Sent Messages.
function rowMouseOver() {
  this.parentNode.className = "rowHover";
}


// onMouseOut handler for rows in Inbox or Sent Messages.
function rowMouseOut() {
  this.parentNode.className = this.parentNode.rowClass;
}
```

You may be wondering about that `rowClass` attribute. There's no such attribute in HTML! It is in fact a custom attribute that's used to store the class of the row that a cell uses. In DOM scripting, you can add attributes at will to nodes, or to HTML tags, which of course are themselves DOM nodes. This comes in very handy when you need to store bits of metadata about a node, as is the case here.

Contrast the two event handler versions, meaning look at the event handlers for the buttons vs. the handlers for the rows, and you will see they are a bit different. Part of it is that they *had* to be different, and part of it is that I *wanted* them to be for the sake of seeing two approaches. First, let's discuss why they had to be different . . . as you can see in Figures 4-3 and 4-4, the Inbox and Sent Messages views contain a table listing messages that you can click to view the message. The contents of these tables are generated dynamically. Part of that generation is attaching event handlers to the cells in the table. As a preview, look at this snippet:

```
td1 = document.createElement("td");
td1.onmouseover = rowmouseover;
```

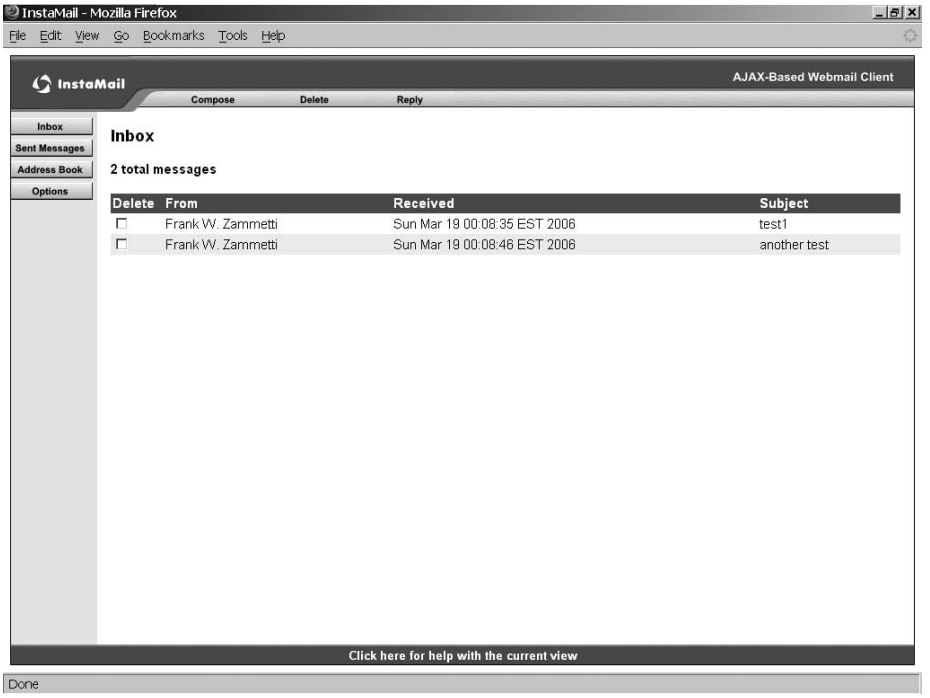


Figure 4-3. *The InstaMail Inbox view*

Figure 4-4. *The InstaMail Sent Messages view*

The first line should be obvious: it is asking the document object to create a new <td> element for us. Note that this element does not immediately become part of the DOM. Instead, it's simply created and returned to the caller. The second line attaches an event handler to the onMouseOver event. Recall that in Javascript, functions are true objects. What we're in effect doing here is setting the onMouseOver property of the new <td> element to the rowMouseOver object, which just so happens to be a function. Note the syntax, though: there is no parameter list given when doing this. The fact is that it's not possible to pass parameters to a handler when you set it on an object like this. Therefore, unlike the button handlers, we could not pass the this reference to it.

Fortunately for us, the this reference is intrinsic when you attach an object to another in this manner. Recall how prototyping works in JavaScript (or read the "Prototype-Based Languages" sidebar if you're not sure) and this should make sense. The onMouseOver property is now an object reference, which for all intents and purposes becomes a part of the object it is attached to, the <td> in this case. So, within this function, this has meaning.

PROTOTYPE-BASED LANGUAGES (NOT TO BE CONFUSED WITH THE PROTOTYPE LIBRARY!)

Any language that is prototype-based, as is JavaScript, uses a style of programming that doesn't use classes like a typical object-oriented language does. Instead, clones of existing objects are made, which endows the extending object to inherit the behaviors of the object serving as the prototype.

For example, let's say you have the following object:

```
function MyFunction() {
  this.a = 1;
  this.b = 2;
}
```

Remember that in JavaScript, functions are first-class citizens and serve as constructors for what you can think of as classes.

Now, if you want a new instance of the MyFunction "class" to work with, you simply do

```
var mf = new MyFunction();
```

If you then did

```
alert(mf.a + mf.b);
```

you would see the expected alert messages saying "3". Further, if you wanted to extend this object pointed to by mf, you could simply do

```
mf.c = 3;
```

Then, when you did the alert:

```
alert(mf.a + mf.b + mf.c);
```

you would again see an alert message, now saying "6" as you'd expect.

So that's why it *had* to be different. As for why I *wanted* it to be different, I wanted to show you that you don't have to change style attributes directly, as we did for the `cursor` attribute of the buttons. Instead, you can just change the class appropriately. If you look at the `rowHover` style selector, you'll see that the `cursor` attribute is set to `pointer`, the same as was done manually for the buttons. This is another example of abstracting out the display properties of your views. With this approach, you do not have to touch the code, just the style sheet. This is generally better because it is less error prone and requires less (or at least different) expertise. Theoretically, nonprogrammers can alter the style sheets themselves, for instance. Think of graphic designers, the people who have the expertise to make it look good rather than make it function.

Typing Too Much Can Be Hazardous to Your Health

Next in `script.js` is a batch of `setXXXX` and `getXXXX` functions. The purpose of these functions is to provide us a shortcut to setting and getting various properties of elements on the page. Consider this line:

```
document.getElementById("myDiv").style.display = "block";
```

We have seen this sort of thing before, and it is certainly not anything scary at this point. However, consider the analogy that is possible because of these functions:

```
setDisplay("myDiv", "block");
```

If nothing else, that's about half the amount of typing! More important, though, it's less error prone and makes the code look quite a bit cleaner and perhaps easier to comprehend. Simply stated, that's the reason for the series of getters and setters you see in the utility functions section.

Following this are a few more utility functions. Most of them simply make calls to the set methods described earlier. For instance, `hideAllLayers()` is called to, well, hide all the `<div>`s for all the views! This is done when a new one is being shown. Instead of needing to hide the currently showing `<div>`, they're all hidden and then the new one is shown. Kind of a lazy approach to it, but it's a bit more foolproof.

Telling Users to Hold Their Horses

After that is `showPleaseWait()`, which is responsible for showing our float-over. This is called pretty much whenever the view changes, or when any operation is undertaken that might take a moment (like saving options or creating a new contact, for example). It begins by calling `hideAllLayers()`, and then calling `enableButtons(false);`, which hides all the buttons on the top and bottom. It would cause problems if the user clicked one of the buttons while an Ajax request was being processed, and the easiest way to avoid that is simply to hide the buttons. Again, this is perhaps a bit of a lazy approach, but is at the same time more foolproof. Once those things are done, `divPleaseWait` is centered on the page, taking into account the current size of the window. The code that accomplishes that is interesting:

```
// First we center the layer.
pleaseWaitDIV = getElement("divPleaseWait");
if (window.innerWidth) {
  lca = window.innerWidth;
} else {
  lca = document.body.clientWidth;
}
lcb = pleaseWaitDIV.offsetWidth;
lcx = (Math.round(lca / 2)) - (Math.round(lcb / 2));
iebody = (document.compatMode &&
  document.compatMode != "BackCompat") ?
  document.documentElement : document.body;
dsocleft = document.all ? iebody.scrollLeft : window.pageXOffset;
pleaseWaitDIV.style.left = (lcx + dsocleft - 120) + "px";
if (window.innerHeight) {
  lca = window.innerHeight;
} else {
  lca = document.body.clientHeight;
}
lcb = pleaseWaitDIV.offsetHeight;
lcy = (Math.round(lca / 2)) - (Math.round(lcb / 2));
iebody = (document.compatMode &&
  document.compatMode != "BackCompat") ?
  document.documentElement : document.body;
dsoctop = document.all ? iebody.scrollTop : window.pageYOffset;
pleaseWaitDIV.style.top = (lcy + dsoctop - 40) + "px";
// Now actually show it.
pleaseWaitDIV.style.display = "block";
```

The thing that makes this a bit tricky is that the values we need to do the calculations involved differ between browsers. Specifically, `innerWidth` is used in IE, while `clientWidth` is used in other browsers, and likewise for `innerHeight` and `clientHeight`. There is also some complication within IE itself in getting a reference to the document body element. There's a difference in the way you gain access to it when you're in strict CSS mode and when you're not. The trinary logic line you see getting a reference to `iebody` is where that logic comes in. At the end of the day though, this all boils down to a relatively simple formula: take the width of the `divPleaseWait` `<div>`, subtract it from the width of the window, and divide by 2. Do the same for the height and the results are your new X and Y coordinates. There's also a need to use some "magic numbers" in the calculations because of various elements of chrome on the browser windows that sometimes are not taken into account. You can consider them a fudge factor, a necessary evil in this case.

Magic numbers are values embedded in code that do not by themselves convey their meaning explicitly. For instance, if you see `a=5*3.14;` in code, you may well reckon from the context of the code that 3.14 is the value of PI, but you may in fact be wrong. In either case, you shouldn't have to reckon anything! It makes for far better code (in terms of clarity and maintainability) to instead create a constant named PI that has the value 3.14 and use that in place of the literal value in the code. Although most programmers worth their weight in Mountain Dew know this, in practice we pretty much universally all break the rule from time to time whether out of laziness or the belief that the value is obvious, or because we believe the value would never change. That doesn't justify it of course, just a truth we should recognize (hopefully you at least comment the code in these situations, as I've failed to do . . . to illustrate this point. Yeah, that's it, that's why! –wink-). As an aside (more so than this note in the first place!) is the origin of the term magic number. It was originally found in some early Unix source code comments and has stuck ever since.

The magic numbers in the code here, by the way, are calculated with an advanced mathematical technique known in academic circles as trial and error . . . I just adjusted them little by little until things looked right in the browser. That's why they're magic numbers in the first place: there's no clear meaning by looking at the code what they are or where they came from.

The `hidePleaseWait()` function is called when an operation completes, that is, when an Ajax request returns. It does as it says: it hides the `divPleaseWait <div>` and displays the buttons again.

The `showView()` function accepts the name of one of the view `<div>`s and shows it. Lastly, the `enableButtons()` function accepts a Boolean value. When `true`, all the buttons on the top and bottom are shown. When `false`, they are hidden.

Next up, going linearly through `script.js`, are all the functions that perform our Ajax. I would like to skip them for now and finish off the non-Ajax functions that follow them. I believe this will help you understand what is going on in the Ajax code a little more easily.

The first non-Ajax function after that section is the `editContact()` function, called when the user clicks one of the existing contacts in the Address Book to edit it:

```
// Called to edit/delete an existing contact.
function editContact() {
  setValue("contactNameEntry", this.parentNode.cells[0].innerHTML);
  setValue("contactAddressEntry", this.parentNode.cells[1].innerHTML);
  setValue("contactNoteEntry", this.parentNode.cells[2].innerHTML);
  setDisabled("contactNameEntry", true);
  setDisabled("contactAddressEntry", false);
  setDisabled("contactNoteEntry", false);
}
```

Incidentally, Figure 4-5 shows you what it looks like when you're editing a contact.

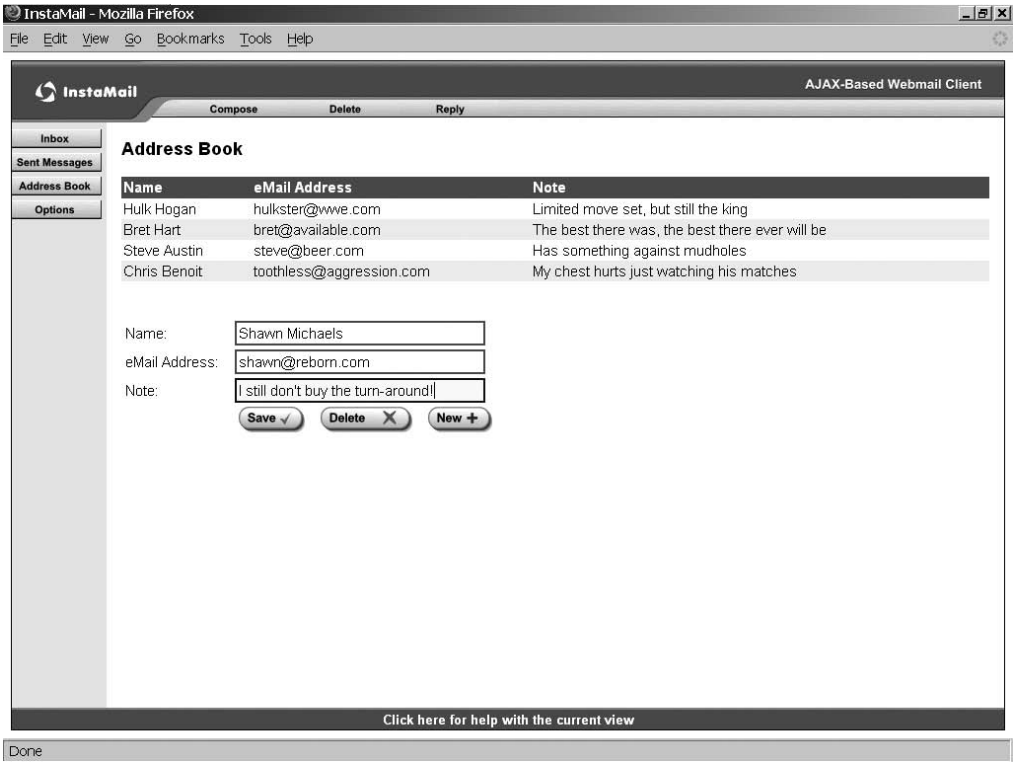


Figure 4-5. *Editing a contact*

Here, and elsewhere, you may have noticed the `this.parentNode` call. This is included due to the structure of the table that is constructed to list messages for either the Inbox or the Sent Messages view. Recall that both views have a check box as the first item on the row. Originally, I had all the `mouseOver`s and `onClick` events on the row, because that's where the information on the message is stored. Besides that, we really are interested in the row, not the individual cells. The problem I encountered, however, is that if you click one of the check boxes in that setup, it fires the `onClick` event of the row! This is certainly not what we want. The way I chose to avoid this is to put the `mouseOver`s and `onClick`s on the individual cells, except for the one containing the check box. That's why if you hover over a row, notice that when you hover over the check box, the row does not highlight. This is perfectly OK and gets around this problem.

However, now when a cell is clicked, the information for the message is needed, and that exists on the row level, not the cell level. Therefore, we take the `this` reference, which is to the cell, and get a reference to its parent node, which is the row. We can then go ahead and access the information we need. In this function, we are setting the values of the edit boxes to the values for the contact the user clicks (the Address Book view has the same sort of table display, a master-detail view, as it is typically called). Again, we face the same problem. The element that generates the `onClick` event is a cell, but we need data on the row; hence we again need to go after the parent node of the cell. In this case, we then need to get the `innerHTML` values of the three cells in the row, which we do by way of an array reference to the array of cells that the parent `<tr>` element has.

Notice too that the contact name is not editable, and we therefore need to disable it. The contact name is, in a sense, the unique key, if this were a database table. Therefore, we do not want the user to be able to change that key, and hence the field is not editable.

Next up is `gotoComposeMessage()`. The only complication here is that if we are in the Address Book view, we assume that the user has selected a contact and wants to compose a message to that person. In that case, we grab the e-mail address and populate the address field on the Compose view. Note that the showing and hiding of the Please Wait `<div>` is pretty much superfluous since the view switch is basically instantaneous. I only do it for the sake of consistency, since anywhere else the view changes, the Please Wait `<div>` is shown. The message composition screen, which incidentally is almost the same as the screen for composing a reply, can be seen in Figure 4-6.

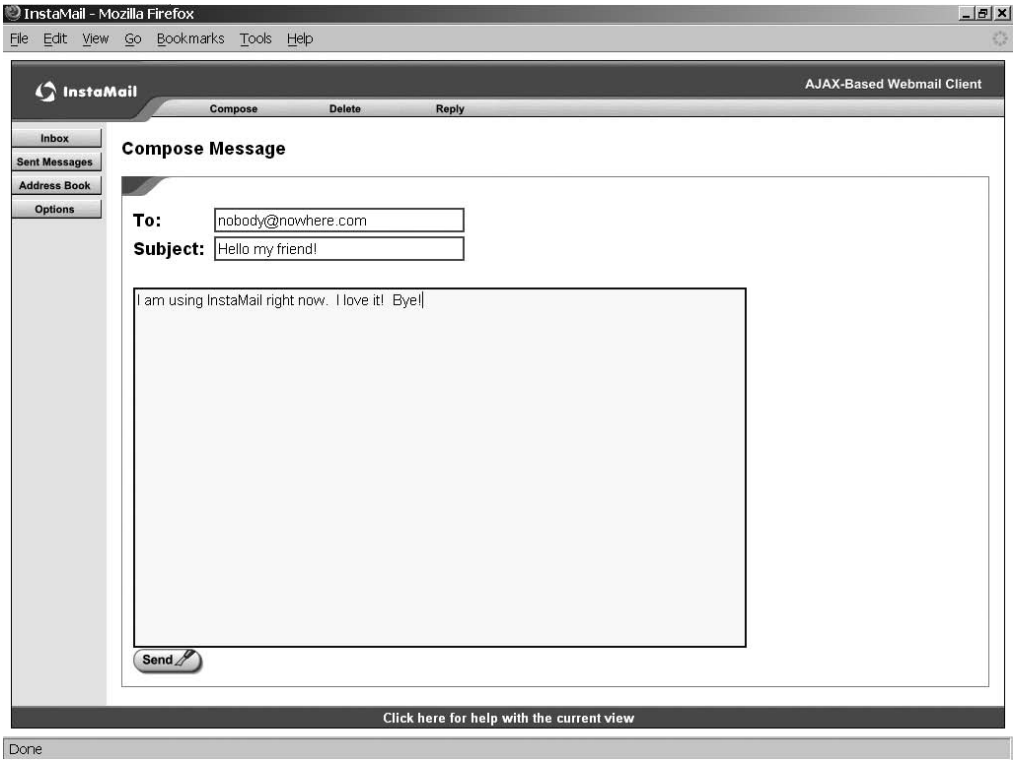


Figure 4-6. *Composing a new message (or a reply to a message)*

`gotoComposeReply()` comes next, and it's very much the same as `gotoComposeMessage()`. However, since the Reply button only has meaning in the Message view, we need to check for that and abort if the user is seeing another view. Beyond that, like `gotoComposeMessage()`, we are prefilling the fields on the Compose view, namely the address of the sender of the message and the subject with "RE:" appended to the front, and the message itself is quoted below a divider line demarcating it from our reply.

The next function we see is `newContact()`. It's almost the same as `editContact()`, except that we're setting the entry fields to blanks, and of course making sure all three edit boxes are now enabled. Otherwise, it is much the same thing.

The last non-Ajax function we encounter is `gotoHelp()`. This is what gets called when the user clicks the help link on the bottom of the page. It's about as simple a help system as one could imagine: based on the current view, it pops a JavaScript `alert()` box with some text describing the current view. Perhaps not the fanciest way to provide online help, but it certainly gets the job done!

Finally, Some Ajax to Look At

Now we'll start to look at the Ajax functions using DWR, which is where all the truly interesting stuff happens. First up is `gotoInbox()`, and its companion function, `replyGetInboxContents()`. You will see this function/companion function dichotomy repeat itself a number of times going forward. You will quickly understand why.

```
// Called when the Inbox link is clicked, shows divInbox.
function gotoInbox() {
  showPleaseWait();
  MailRetriever.getInboxContents(replyGetInboxContents);
}
// Our callback.
var replyGetInboxContents = function(data) {
  DWRUtil.removeAllRows("inboxTBody");
  checkboxNumber = 0;
  var altRow = true;
  var getFrom = function(data) { return data.from; };
  var getReceived = function(data) { return data.received; };
  var getSubject = function(data) { return data.subject; };
  var getCheckbox = function(data) {
    var cb = document.createElement("input");
    cb.type = "checkbox";
    cb.id = "cb_received_" + checkboxNumber;
    cb.msgID = data.msgID;
    checkboxNumber++;
    return cb;
  };
  var count = 0;
  DWRUtil.addRows("inboxTBody", data,
    [ getCheckbox, getFrom, getReceived, getSubject ], {
    rowCreator:function(options) {
      count++;
      var row = document.createElement("tr");
      if (altRow) {
        row.rowClass = "msgListRow1";
        row.className = "msgListRow1";
        altRow = false;
      } else {
        row.rowClass = "msgListRow2";
        row.className = "msgListRow2";
        altRow = true;
```

```
      }
      row.msgType = data[options.rowIndex].msgType;
      row.msgID = data[options.rowIndex].msgID;
      return row;
    },
    cellCreator:function(options) {
      var cell = document.createElement("td");
      if (options.cellNum != 0) {
        cell.onclick = gotoViewMessage;
        cell.onmouseover = rowMouseOver;
        cell.onmouseout = rowMouseOut;
      }
      return cell;
    }
  });
  setInnerHTML("inboxCount", count + " total messages");
  currentView = "divInbox";
  showView(currentView);
  hidePleaseWait();
}
```

This function is called, as you might expect, when the Inbox button is clicked. From this point on you will notice that all of the Ajax functions have a similar pattern, that is, there is one function that is called in response to some event (usually user initiated), and then an associated function. The associated function is akin to the Ajax callback function you attach to an `XMLHttpRequest` object. DWR hides all the details of Ajax from you, but it still is the same underlying principle.

In this case, we simply show the Please Wait float-over, and then this statement is executed:

```
MailRetriever.getInboxContents(replyGetInboxContents);
```

As previously discussed, DWR generates a JavaScript file for each object you want to remote. This JavaScript contains essentially "stub" objects that you call on, and DWR behind the scenes makes the remote call to the server-side object. This gives you the illusion of directly calling remote objects. That's precisely what this line is doing. It's calling the `getInboxContents()` method of the `MailRetriever` object, or more precisely, of a new `MailRetriever` object because a new object is created as a result of each call, so there is not just one object to call. The only parameter we pass in this case is the JavaScript callback function that will handle the return from the method call. This parameter is always present in all the DWR code in this project (and indeed in most of the DWR code throughout this book), and it is always the last parameter.

When the call returns, the `replyGetInboxContents()` function is called. Notice the somewhat unusual way the function is defined: it is assigned to a variable! This gives us an easy way to refer to the function. You can in fact have the callback function inline in the method call, that is, something along the lines of

```
MailRetriever.getInboxContents(function replyGetInboxContents(str){alert(str);});
```

Most people, myself included, believe that is just simply harder to read, and thus I try and avoid that where possible. The other problem is that should you want to share the callback with another Ajax call, you will not be able to do it if the function is inlined because it is essentially an anonymous function.

DWR: It's Not Just for Ajax Any More

Once we get into the callback function, the first thing we see is

```
DWRUtil.removeAllRows("inboxTBody");
```

`DWRUtil` is an object that DWR provides that is completely optional; it is not required for DWR to work. However, it provides some very handy functions, one of which is this `removeAllRows()` function. Its purpose is to remove all the rows from a given table (`<tbody>` more precisely). This is what we need to do here. When the Inbox is refreshed, we need to start with an empty table, and then create a row for each e-mail in the Inbox. This single line accomplishes that first part, clearing the table.

After that comes the following chunk of code:

```
checkboxNumber = 0;
var altRow = true;
var getFrom = function(data) { return data.from; };
var getReceived = function(data) { return data.received; };
var getSubject = function(data) { return data.subject; };
var getCheckbox = function(data) {
var cb = document.createElement("input");
cb.type = "checkbox";
cb.id = "cb_received_" + checkboxNumber;
cb.msgID = data.msgID;
checkboxNumber++;
return cb;
};
var count = 0;
```

Recall our discussion of the `checkboxNumber` variable previously. This is used to append a number to the ID of each delete check box we'll create, so that they all have a unique ID. Numbering starts at 0, hence this line. The variable `altRow` is a simple Boolean toggle flag that we'll use to set the appropriate class for each row in the table so that we get the alternating-band effect we want.

The next four functions are functions that are required by DWR, as we'll see shortly. Each one is analogous to an accessor (or getter) method on a javabean. In this case, DWR will call these functions and pass them an object referenced by the parameter `data`. The functions will access some property of that data, just as if it were a javabean, and return it. Each of these functions corresponds to a cell in the table. This will become clear in a moment! The last function, the `getCheckbox()` function, will be called by DWR to create the delete check box for each row.

Notice the use of the `checkboxNumber` variable. Also notice that the only parameter to this function is the `data` variable, which is created by DWR. It should now make sense why `checkboxNumber` has to be global: there is no other way to get it to this function! Since its value has to persist between invocations of this function, the variable has to exist outside the function.

After this comes a call to another very useful function from the DWRUtil object:

```
DWRUtil.addRows("inboxTBody", data,
  [ getCheckbox, getFrom, getReceived, getSubject ], {
  rowCreator:function(options) {
    count++;
    var row = document.createElement("tr");
    if (altRow) {
      row.rowClass = "msgListRow1";
      row.className = "msgListRow1";
      altRow = false;
    } else {
      row.rowClass = "msgListRow2";
      row.className = "msgListRow2";
      altRow = true;
    }
    row.msgType = data[options.rowIndex].msgType;
    row.msgID = data[options.rowIndex].msgID;
    return row;
  },
  cellCreator:function(options) {
    var cell = document.createElement("td");
    if (options.cellNum != 0) {
      cell.onclick = gotoViewMessage;
      cell.onmouseover = rowMouseOver;
      cell.onmouseout = rowMouseOut;
    }
    return cell;
  }
});
```

addRows() is another table modification function, this one focused on creating rows. Its first argument is the ID of a <tbody> element to populate. The second argument is the data object passed to the callback, which is the return from the method call on the server-side object. In this case, as is logical for a table, it's a collection, where each element corresponds to a row. After that comes an array of function calls, and they should look very familiar to you!

As I mentioned, each one corresponds to a cell in the table. So, here's what DWR is going to do:

1. Iterate over the data collection. For each element, create a <tr>.
2. Iterate over the collection of functions named. For each, create a <td>, call the function, and populate the <td> with the return from the function.
3. Add the <td> to the <tr>.
4. Add the <tr> to the table.
5. Continue this process until the data collection is exhausted.

The other parameter to the `addRows()` function is optional and is an array of functions that DWR will use to create each `<tr>` and `<td>`. The default implementations simply create a plain `<tr>` and `<td>` and return it, but the designers of DWR wisely recognized that many times you will need to do more than that, so they allow you to provide custom functions for creating these elements. In this case, I broke my own rule and made them anonymous inline functions. I did this more to demonstrate what that looks like, but since they are not going to be shared, it's not such a bad thing, just this once. Still, I think you'll agree that the syntax is a little funky perhaps?

Be that as it may, the first inline function deals with creating a row. We have a couple of considerations to take into account. First, we need to count how many rows there are, which correspond to e-mails, remember, so that we can display how many e-mails are in the Inbox for the user. So, with each invocation, we increment the variable `count`. Second, we need to set the appropriate class for the row to get the banding effect we want. This is done by looking at whether `altRow` is `true` or `false`, and setting a different class accordingly, and of course flipping the value of `altRow`. Third, we need to attach some custom attributes to the `<tr>` that will allow us to identify the e-mail. In this case that means setting `msgType` to the `msgType` the server returns (which will always be "received" in the Inbox) and the `msgID` returned by the server (which is just the number of the message in the Inbox). Notice the array notation used to get that information. DWR passes us an options object, which includes the data object originally passed in to the callback. There doesn't seem to be a way to directly access the current row, that is, there does not appear to be something like `options.currentRow` available. However, we do have access to the index into the data array that is currently being rendered. In other words, we can tell what row we are working on. So, with that, we can use array notation against the data variable to get the information we need.

The second inline function deals with creating a cell. Recall previously I described the problem with putting event handlers on the row when the user clicks the delete check box. I mentioned that the handlers instead had to be attached to the cells. This is where that is done.

The function first checks whether this is the first cell being added, and if it is, then it is the cell with the check box, so no event handlers are attached. If it is any other cell, the `onClick`, `onMouseOver`, and `onMouseOut` events are hooked. That is all this function does.

Lastly, in our `gotoInbox()` function we find

```
setInnerHTML("inboxCount", count + " total messages");
currentView = "divInbox";
showView(currentView);
hidePleaseWait();
```

The first line uses one of our convenience functions to display for the user how many e-mails are in the Inbox, again using that `count` variable we saw before. The second line sets the view to the Inbox, which it may or may not have been set to already. The third line actually shows the view, and finally we hide the Please Wait float-over, and the function is done.

DWR, in addition to making Ajax and RPC very easy, also makes table manipulation very easy! Since this is a very common thing to do with Ajax, it is a very nice feature indeed.

Next up is the `gotoSentMessage()` function. This is called when the Sent Messages button is clicked. I am going to refrain from going over it because it is virtually identical to the `gotoInbox()` function we just looked at. The only real difference is that sent messages are identified by their file names as opposed to their `msgIDs`, as messages in the Inbox are. Aside from that, it is almost identical code. Do review it, though, to convince yourself of that, and to reinforce how DWR is used.

Viewing Messages

After that comes this function:

```
// Called when a message is clicked while divInbox or divSentMessages is
// showing, shows divMessage.
function gotoViewMessage() {
  var _msgID = null;
  var _msgType = null;
  var _filename = null;
  showPleaseWait();
  if (currentView == "divInbox") {
    _msgID = this.parentNode.msgID;
    _filename = null;
  }
  if (currentView == "divSentMessages") {
    _filename  = this.parentNode.filename;
    _msgID = null;
  }
  _msgType = this.parentNode.msgType;
  MailRetriever.retrieveMessage(_msgType, _filename, _msgID, replyRetrieveMessage);
}
// Our callback.
var replyRetrieveMessage = function(data) {
  setInnerHTML("msgSubject", data.subject);
  setInnerHTML("msgText", data.msgText);
  setInnerHTML("msgType", data.msgType);
  if (data.msgType == "received") {
    setInnerHTML("msgID", data.msgID);
    setInnerHTML("msgFromToLabel", "From: ");
    setInnerHTML("msgFromTo", data.from);
    setInnerHTML("viewTitle", "Received Message");
    setInnerHTML("msgSentReceivedLabel", "Received: ");
    setInnerHTML("msgSentReceived", data.received);
  }
  if (data.msgType == "sent") {
    setInnerHTML("msgFilename", data.filename);
    setInnerHTML("msgFromToLabel", "To: ");
    setInnerHTML("msgFromTo", data.to);
    setInnerHTML("viewTitle", "Sent Message");
    setInnerHTML("msgSentReceivedLabel", "Sent: ");
    setInnerHTML("msgSentReceived", data.sent);
  }
  currentView = "divMessage";
  showView(currentView);
  hidePleaseWait();
}
```

Review either the `gotoInbox()` or `gotoSentMessages()` function, and you will see that the `gotoViewMessage()` function is in fact the `onClick` handler set for each cell. This function has a relatively simple job: get the details we need to identify an e-mail, which means the message type ("sent" or "received") and either the `msgID` or `filename`, whichever is appropriate. We have discussed why the cells have to be the trigger of the event, and we also discussed the use of the `parentNode` property to get at the information we need, and that is all that is going on here.

Once the information is retrieved, a call is made to the `retrieveMethod()` method of the `MailRetriever` object on the server.

The callback takes the information retrieved from the server about the message and populates various fields on the screen, then shows the message view (see Figure 4-7). Based on whether this is a sent message or a received message (remember, this function services both the Inbox view and the Sent Messages view), not only does the information have to be displayed, but we also need to update various labels on the screen. This callback is really quite straightforward.

Figure 4-7. *Viewing a message, a sent one in this case*

Deleting Messages

The `doDelete()` function is next, called when the Delete button is clicked. The first thing it does is check the value of the `currentView` variable, and if it is not the Message view, the Inbox view, or the Sent Messages view, it tells the user it doesn't apply.

When the view is the Message view, it's a fairly straightforward operation. All `doDelete()` has to do is grab the `msgType`, `filename`, and `msgID` values from the `<div>`s that data is in. Only the `filename` or the `msgID` is important, not both (i.e., when viewing the Inbox, the `msgID` applies; when viewing Sent Messages, then `filename` applies), but it does no harm to send both since the server-side code looks at `msgType` and only cares about one or the other based on that. Once that is done, `deleteMessages()` of the `MailDeleter` object is called. The response handler simply looks at the current view and refreshes it. That means calling `gotoInbox()` if viewing the Inbox, which results in a new Ajax request, or `gotoSentMessage()` if viewing the Sent Messages (again resulting in another Ajax request). If the current view was the Message view, it examines the value of the title on the page to determine whether a received message or a sent message was being viewed, and then calls `gotoInbox()` or `gotoSentMessages()` as appropriate.

Oops, I got ahead of things a bit there! What happens if the current view is the Inbox? In that case, we are dealing with the possibility of the user having checked off a number of e-mails to delete. The `deleteMessages()` method of the `MailDeleter` class actually accepts a comma-separated list of values, either `filenames` or `msgID`s, depending on the view. When the view was the individual message view, there was just a single value. This is still a valid comma-separated list, of course; it just happens to have only a single value. For the other two views, though, we have to construct a list of all the checked items. To do so, we use this code:

```
var obj = null;
var i = 0;
// Construct a CSV list of MsgIDs for Inbox messages.
if (currentView == "divInbox") {
  msgType = "received";
  filenames = null;
  obj = getElement("cb_received_" + i);
  while (obj != null) {
    if (obj.checked) {
      if (msgIDs != "") {
        msgIDs += ",";
      }
      msgIDs += obj.msgID;
    }
    i++;
    obj = getElement("cb_received_" + i);
  }
}
```

Recall when we populated the table, we gave each check box a unique ID that ended with a number. Now you can see why: it facilitates us easily cycling through all the check boxes looking for those that are checked. Of course, we don't know how many messages there are, and thus how many check boxes there are (true, we could examine the value of the `<div>` where the message count is displayed, but that seems like a bit of a hack to me). So, we get an object reference to the first check box. Assuming we do not get null back, indicating the check box doesn't exist (there are no messages in this view), then we see whether it is checked and add it to our CSV list if so. We then try and get the next check box, and so on, building up our

CSV string. Once we are done, the same process as for a single message occurs in terms of sending the request and refreshing the correct view when the call returns.

Sending Messages

We now come to the `sendMessage()` function. This is relatively simple. First, we perform a check to be sure the user entered an e-mail address to send to, a subject, and some text, and if not we tell the user we cannot yet send the message. Assuming the user has entered all three, the `sendMessage()` method of the `MailSender` class is called. Notice the call in this case contains parameters:

```
MailSender.sendMessage(composeTo, composeSubject, composeText,
replySendMessage);
```

The last argument is still the callback function, but anything preceding that is taken by DWR to be arguments to the target method.

The response handler clears out the entry boxes and shows the Send Messages view. Sending a message couldn't be easier!

Dealing with Contacts and the Address Book

After that is the `gotoAddressBook()` function. This, like `gotoSentMessages()`, is very similar to the `gotoInbox()` function that we have examined, so please have a look; it will be very familiar! The only real difference is that because we do not have delete check boxes in this view, there is no check when creating a cell to see which cell number we are creating. In fact, because there are no check boxes, I could have put the event handlers on the row this time. I did not do so just for the sake of consistency with the other functions.

`saveContact()` comes next, and there really is not much to it:

```
// Called to save a contact.
function saveContact() {
  contactName = getValue("contactNameEntry");
  contactAddress = getValue("contactAddressEntry");
  contactNote = getValue("contactNoteEntry");
  if (contactName == "" || contactAddress == "") {
    alert("Please enter both a name and address");
    hidePleaseWait();
    return false;
  }
  showPleaseWait();
  AddressBookManager.saveContact(contactName, contactAddress, contactNote,
    replySaveContact);
}
// Our callback.
var replySaveContact = function(data) {
  newContact();
  alert(data);
  hidePleaseWait();
  gotoAddressBook();
}
```

We once again have the typical two cooperative functions. A quick check to be sure the user entered all the required information, and then it is just a matter of an Ajax call to the `AddressBookManager`'s `saveContact()` method, passing the appropriate arguments along. The server will respond with a success or failure message, so all the response handler has to do is display what was returned. This is akin to doing a `System.out.println` on the return from a method on the server, since `saveContact()` returns a plain old string. One problem I encountered is that there is sometimes a slight delay before the Address Book file is actually written to disk (this is also true when deleting a contact). Because of this, the returned message informs users that they may have to click the Address Book link to see the new contact.

To go along with `saveContact()` is `deleteContact()`:

```
// Called to delete a contact.
function deleteContact() {
  contactName = getValue("contactNameEntry");
  if (contactName == "") {
    alert("Please select a contact to delete");
    hidePleaseWait();
    return false;
  }
  showPleaseWait();
  AddressBookManager.deleteContact(contactName, replyDeleteContact);
}
// Our callback.
var replyDeleteContact = function(data) {
  newContact();
  alert(data);
  hidePleaseWait();
  gotoAddressBook();
}
```

Very much like `saveContact()`, `deleteContact()` checks for the required information, and then makes an Ajax call to perform the delete. The response handler again simply displays the returned string and shows the Address Book again.

Configuring Options-Related Code

We are almost done exploring the client-side code; just two functions left (well, really four, but I have been counting the pairs as one in essence). Next we have `gotoOptions()`, called when the Options button is clicked:

```
// Called when the Options link is clicked, shows divOptions.
function gotoOptions() {
  showPleaseWait();
  OptionsManager.retrieveOptions(replyRetrieveOptions);
}
// Our callback.
var replyRetrieveOptions = function(data) {
  setValue("pop3ServerEntry", data.pop3Server);
  if (data.pop3ServerRequiresLogin == "true") {
```

```
    setChecked("pop3ServerRequiresLoginEntry", "true");
  } else {
    setChecked("pop3ServerRequiresLoginEntry", null);
  }
  setValue("pop3UsernameEntry", data.pop3Username);
  setValue("pop3PasswordEntry", data.pop3Password);
  setValue("smtpServerEntry", data.smtpServer);
  if (data.smtpServerRequiresLogin == "true") {
    setChecked("smtpServerRequiresLoginEntry", "true");
  } else {
    setChecked("smtpServerRequiresLoginEntry", null);
  }
  setValue("smtpUsernameEntry", data.smtpUsername);
  setValue("smtpPasswordEntry", data.smtpPassword);
  setValue("fromAddressEntry", data.fromAddress);
  currentView = "divOptions";
  showView(currentView);
  hidePleaseWait();
}
```

There really is not much to this. A call is made to the server, which returns the options, and the response handler then populates the fields of the Options view with the data, which can be seen in Figure 4-8.

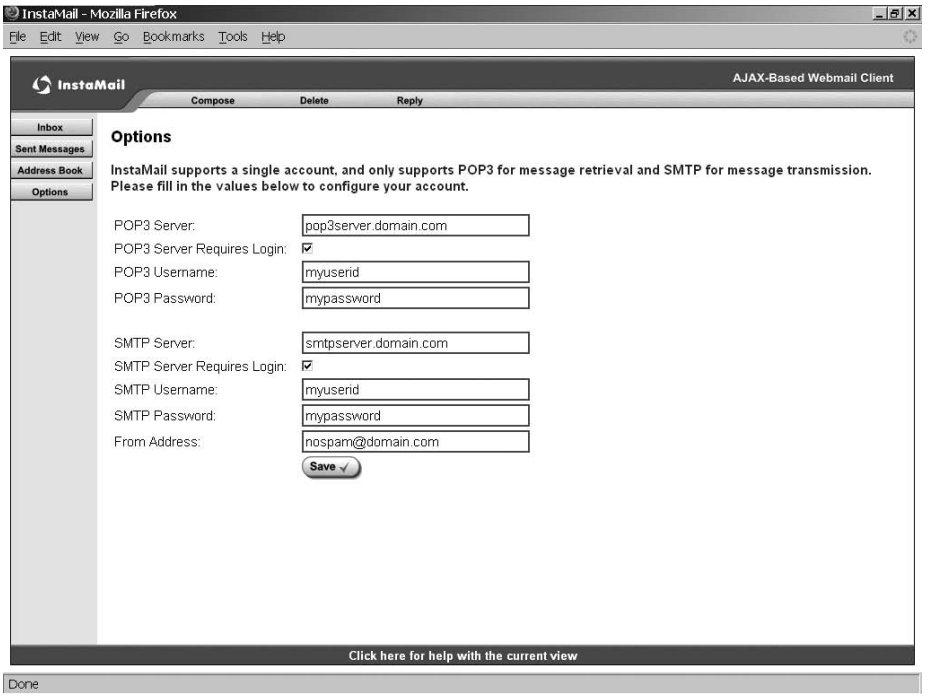


Figure 4-8. *The InstaMail Options screen*

One thing I would like to bring to your attention is that the call to `retrieveOptions()` returns an `OptionsDTO`, as we'll see when we look at the server-side code. Recall the `dwr.xml` configuration file, in which we allowed DWR to convert the `OptionsDTO`. DWR is able to essentially take that bean and convert it to a JavaScript object (that's effectively what happens anyway, whether that's really what happens is a DWR detail we don't so much care about for our purposes here). What is important, however, is that DWR is giving us the *illusion* of having that `OptionsDTO` on the client and allowing us to access its fields via the `data` variable. This is pretty cool! The cleanliness of the previous code in terms of actually accessing the options definitely is.

And now finally, we reach the final client-side code, and I am sure you can guess what it is: yep, it is `saveOptions()`!

```
// Called when the Save button is clicked when divOptions is showing.
function saveOptions() {
  pop3Server = getValue("pop3ServerEntry");
  pop3ServerRequiresLogin = getChecked("pop3ServerRequiresLoginEntry");
  pop3Username = getValue("pop3UsernameEntry");
  pop3Password = getValue("pop3PasswordEntry");
  smtpServer = getValue("smtpServerEntry");
  smtpServerRequiresLogin = getChecked("smtpServerRequiresLoginEntry");
  smtpUsername = getValue("smtpUsernameEntry");
  smtpPassword = getValue("smtpPasswordEntry");
  fromAddress = getValue("fromAddressEntry");
  if (pop3Server == "") {
    alert("You must enter a POP3 server address");
    hidePleaseWait();
    return false;
  }
  if (pop3ServerRequiresLogin == true &&
    (pop3Username == "" || pop3Password == "")) {
    alert("If the POP3 server requires login, then you must enter " +
      "both a POP3 username and password");
    hidePleaseWait();
    return false;
  }
  if (smtpServer == "") {
    alert("You must enter an SMTP server address");
    hidePleaseWait();
    return false;
  }
  if (smtpServerRequiresLogin == true &&
    (smtpUsername == "" || smtpPassword == "")) {
    alert("If the SMTP server requires login, then you must enter " +
      "both an SMTP username and password");
    hidePleaseWait();
    return false;
  }
  if (fromAddress == "") {
```

```
    alert("You must enter a from address");
    hidePleaseWait();
    return false;
  }
  showPleaseWait();
  OptionsManager.saveOptions(pop3Server, pop3ServerRequiresLogin,
    pop3Username, pop3Password, smtpServer, smtpServerRequiresLogin,
    smtpUsername, smtpPassword, fromAddress, replySaveOptions);
}
// Our callback.
var replySaveOptions = function(data) {
  alert(data);
  appConfigured = true;
  setClassName("divGettingStarted", "divHidden");
  showView(currentView);
  hidePleaseWait();
}
```

First, the options that the user entered are grabbed. Next, a series of validations are performed to be sure they entered all required information. Once that is done, the Ajax call is made to `saveOptions()` on the `OptionsManager` class, and all the options that were entered are passed along. The callback has only to display the result, again a simple string, and reshow the Options view.

The Server-Side Code

We are now ready to move on to the server-side code, which is encapsulated in a total of eight classes. Three of them are simple DTOs, nothing but data fields and their getter and setter methods. I will leave them to you to examine, but there is virtually no code to look at, save for the overridden `toString()` method (it's my standard reflection-based method—do feel free to steal it!).

OptionsManager.java

The first class to look at is the `OptionsManager` class, shown in Listing 4-3 and in the UML diagram in Figure 4-9.

Listing 4-3. *The* `OptionsManager` *Class Code*

```
package com.apress.dwrprojects.instamail;


import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.util.Properties;
import javax.servlet.ServletContext;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;


/**
 * This class deals with maintaining options, including the e-mail account.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>.
 */
public class OptionsManager {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(OptionsManager.class);


  /**
   * File name of the options file.
   */
  private static final String optionsFilename = "options.properties";


  /**
   * This method retrieves the options and returns them.  If no
   * optionsFilename file is found, a 'blank' DTO is returned.
   *
   * @param  sc ServletContext associates with the request.
   * @return    An OptionsDTO containing all the stored options.
   */
  public OptionsDTO retrieveOptions(ServletContext sc) {

    // Instantiate an OptionsDTO, and by default assume it will be configured.
    // This means the application has already been configured for use.  This
    // affects what the user can do when the app is accessed initially.
    OptionsDTO options = new OptionsDTO();
    options.setConfigured(true);

    // Read in the options.
    InputStream isFeedFile =
      sc.getResourceAsStream("/WEB-INF/" + optionsFilename);
    Properties props = new Properties();
    try {
      if (isFeedFile == null) {
        throw new IOException(optionsFilename + " not found");
      }
```

```
      props.load(isFeedFile);
      isFeedFile.close();
    } catch (IOException e) {
      log.info("No " + optionsFilename + " file, a blank DTO will " +
        "be returned.");
      // Make sure the OptionsDTO is set as unconfigured so that when the
      // index.jsp page is loaded, all the user will be allowed to do is go to
      // the Options views.
      options.setConfigured(false);
      props.setProperty("pop3Server", "");
      props.setProperty("pop3ServerRequiresLogin", "false");
      props.setProperty("pop3Username", "");
      props.setProperty("pop3Password", "");
      props.setProperty("smtpServer", "");
      props.setProperty("smtpServerRequiresLogin", "false");
      props.setProperty("smtpUsername", "");
      props.setProperty("smtpPassword", "");
      props.setProperty("fromAddress", "");
    }

    // Populate OptionsDTO from options Properties.
    options.setPop3Server(props.getProperty("pop3Server"));
    options.setPop3ServerRequiresLogin(
      props.getProperty("pop3ServerRequiresLogin"));
    options.setPop3Username(props.getProperty("pop3Username"));
    options.setPop3Password(props.getProperty("pop3Password"));
    options.setSmtpServer(props.getProperty("smtpServer"));
    options.setSmtpServerRequiresLogin(
      props.getProperty("smtpServerRequiresLogin"));
    options.setSmtpUsername(props.getProperty("smtpUsername"));
    options.setSmtpPassword(props.getProperty("smtpPassword"));
    options.setFromAddress(props.getProperty("fromAddress"));

    return options;

  } // End retrieveOptions().


  /**
   * This method saves the options.
   *
   * @param  pop3Server              The POP3 server address.
   * @param  pop3ServerRequiresLogin Does the POP3 server require login?
   * @param  pop3Username            The POP3 username.
   * @param  pop3Password            The POP3 password.
   * @param  smtpServer              The SMTP server address.
   * @param  smtpServerRequiresLogin Does the SMTP server require login?
```

```
 * @param  smtpUsername            The SMTP username.
 * @param  smtpPassword            The SMTP password.
 * @param  fromAddress             From address for outgoing messages.
 * @param  sc                      ServletContext associated with the request.
 * @return                         A message saying the save was OK.
 */
public String saveOptions(String pop3Server, String pop3ServerRequiresLogin,
  String pop3Username, String pop3Password, String smtpServer,
  String smtpServerRequiresLogin, String smtpUsername,
  String smtpPassword, String fromAddress, ServletContext sc) {

    // Log what we received.
    log.info("\nSaving options:\n" +
    "pop3Server = " + pop3Server + "\n" +
      "pop3ServerRequiresLogin = " + pop3ServerRequiresLogin + "\n" +
      "pop3Username = " + pop3Username + "\n" +
      "pop3Password = " + pop3Password + "\n" +
      "smtpServer = " + smtpServer + "\n" +
      "smtpServerRequiresLogin = " + smtpServerRequiresLogin + "\n" +
      "smtpUsername = " + smtpUsername + "\n" +
      "smtpPassword = " + smtpPassword + "\n" +
      "fromAddress = " + fromAddress + "\n");

    String result = "";

    // Populate Properties structure.
    Properties props = new Properties();
    props.setProperty("pop3Server", pop3Server);
    props.setProperty("pop3ServerRequiresLogin",
      pop3ServerRequiresLogin);
    props.setProperty("pop3Username", pop3Username);
    props.setProperty("pop3Password", pop3Password);
    props.setProperty("smtpServer", smtpServer);
    props.setProperty("smtpServerRequiresLogin",
      smtpServerRequiresLogin);
    props.setProperty("smtpUsername", smtpUsername);
    props.setProperty("smtpPassword", smtpPassword);
    props.setProperty("fromAddress",  fromAddress);

    // Lastly, delete any existing optionsFilename file in WEB-INF and
    // write out a new version from the Properties object we just populated.
    // Return a message saying the operation was complete, or if any problems
    // occur, a message saying what went wrong.
    FileOutputStream fos = null;
    try {
      new File(sc.getRealPath("WEB-INF") + "/" + optionsFilename).delete();
      fos = new FileOutputStream(sc.getRealPath("WEB-INF") +
```

```
          "/" + optionsFilename);
        props.store(fos, null);
        fos.flush();
        result = "Options have been saved.";
      } catch (IOException e) {
        log.error("Error saving contact:");
        e.printStackTrace();
        result = "Options could not be saved.  " +
          "Please review logs for details.";
      } finally {
        try {
          if (fos != null) {
            fos.close();
          }
        } catch (IOException e) {
          log.error("Error closing fos: " + e);
        }
      }

      return result;

  } // End saveOptions().


} // End class.
```

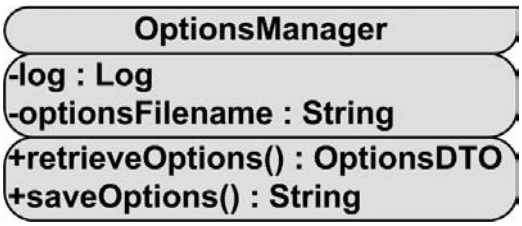


Figure 4-9. *UML diagram of the* `OptionsManager` *class*

This class has two static members, a Commons Logging `Log` instance, and a `String` that stores the name of the file where options will be saved.

The first method we find is `retrieveOptions()`, which, as its name implies, reads the options file and returns an `OptionsDTO` instance populated with the options. If the options file is not found—which most likely means this is the first time the user has used InstaMail—we throw an `IOException`, which is immediately caught, and the `OptionsDTO` is blanked out and returned. You may ask why the exception is thrown and immediately caught. Simply put, it allows the code to blank out the DTO to be in one place and handle not only the known situation of the file not being present (which is not a trigger of an exception by itself), but also any unexpected exceptions that might occur. I usually do not like using exceptions for flow control, which is more or less what this is, because exceptions incur a relatively high amount of

overhead within the JVM to instantiate and throw. In this case, though, I felt it was acceptable because this is not a situation that should arise more than once, and it makes for more concise code.

The options are stored in the form of a standard Java properties file, so we wind up with a populated `Properties` object if the file is actually present. The code then takes the values from that structure and transfers it to the `OptionsDTO` and returns it.

Going along with `retrieveOptions()` is `saveOptions()`. This is even easier! We simply grab the incoming request parameters, which are our options; populate a `Properties` object from it; and then write it out using pretty typical code for writing out a properties file. A string is returned that will be displayed to the user indicating success or failure. Not much to it.

AddressBookManager.java

Now we'll have a look at the `AddressBookManager` class, as shown in Listing 4-4 and in the UML diagram in Figure 4-10.

Listing 4-4. *The* `AddressBookManager` *Class Code*

```
package com.apress.dwrprojects.instamail;


import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.Properties;
import javax.servlet.ServletContext;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;


/**
 * This class deals with maintaining the Address Book.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>.
 */
public class AddressBookManager {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(AddressBookManager.class);
```

```
/**
 * File name of the Address Book.
 */
private static final String addrBookFilename = "addrbook.properties";


/**
 * This method retrieves the contents of the Address Book.
 *
 * @param  sc ServletContext associates with the request.
 * @return    A collection of ContactDTOs.
 */
public Collection retrieveContacts(ServletContext sc) {

  // Read in the Address Book.
  InputStream isFeedFile =
    sc.getResourceAsStream("/WEB-INF/" + addrBookFilename);
  Properties props       = new Properties();
  int        numContacts = 0;
  try {
    if (isFeedFile == null) {
      throw new IOException(addrBookFilename + " not found");
    }
    props.load(isFeedFile);
    isFeedFile.close();
    // Now we need to determine how many contacts there are.  To do this, we
    // divide the total number of properties read in by 3, since each
    // contact always has 3 items stored about them.
    if (props.size() != 0) {
      numContacts = props.size() / 3;
    }
  } catch (IOException e) {
    log.info("No " + addrBookFilename + " file, an empty address book will " +
      "be returned.");
  }

  // Now we cycle through the properties the number of times we calculated
  // there are contacts.  For each we construct a ContactDTO and add it to
  // the collection to be returned.
  log.info("numContacts = " + numContacts);
  Collection<ContactDTO> contacts = new ArrayList<ContactDTO>();
  for (int i = 1; i < numContacts + 1; i++) {
    ContactDTO contact = new ContactDTO();
    contact.setName(props.getProperty("name" + i));
    contact.setAddress(props.getProperty("address" + i));
    contact.setNote(props.getProperty("note" + i));
    contacts.add(contact);
  }
```

```
    return contacts;

  } // End retrieveContacts().


  /**
   * This method adds a contact to the Address Book.
   *
   * @param  inName    The name of the contact.
   * @param  inAddress The e-mail address for the contact.
   * @param  inNote    Any arbitrary note about the contact.
   * @param  sc        ServletContext associates with the request.
   * @return           A message saying the save was OK.
   */
  public String saveContact(String inName, String inAddress,
    String inNote, ServletContext sc) {

      // Log what we received.
      log.info("\nAdding contact:\n" +
        "inName = " + inName + "\n" +
        "inAddress = " + inAddress + "\n" +
        "inNote = " + inNote + "\n");

      String result = "";

      // In order to save a contact, we essentially need to write out the
      // entire Address Book, so the first step is to read it in.
      Collection contacts = retrieveContacts(sc);

      // Now we iterate over it, adding each to a Properties object.  Remember
      // that for each contact we are writing out three properties, the name,
      // the address, and the note.  To name each uniquely, we append a number
      // onto it.
      Properties props = new Properties();
      int i = 1;
      for (Iterator it = contacts.iterator(); it.hasNext();) {
        ContactDTO contact = (ContactDTO)it.next();
        props.setProperty("name"    + i, contact.getName());
        props.setProperty("address" + i, contact.getAddress());
        props.setProperty("note"    + i, contact.getNote());
        i++;
      }

      // Now we add the new contact
      props.setProperty("name"    + i, inName);
      props.setProperty("address" + i, inAddress);
      props.setProperty("note"    + i, inNote);
```

```
    // Lastly, delete any existing addrBookFilename file in WEB-INF and
    // write out a new version from the Properties object we just populated.
    // Return a message saying the operation was complete, or if any problems
    // occur, a message saying what went wrong.
    FileOutputStream fos = null;
    try {
      new File(sc.getRealPath("WEB-INF") + "/" + addrBookFilename).delete();
      fos = new FileOutputStream(sc.getRealPath("WEB-INF") +
        "/" + addrBookFilename);
      props.store(fos, null);
      fos.flush();
      fos.close();
      result = "Contact has been added.\n\nPlease note that if the contact " +
        "does not show up immediately, you may have to click the " +
        "Address Book link once or twice.";
    } catch (IOException e) {
      log.error("Error saving contact:");
      e.printStackTrace();
      result = "Contact could not be added.  Please review logs for details.";
    } finally {
      try {
        if (fos != null) {
          fos.close();
        }
      } catch (IOException e) {
        log.error("Error closing fos: " + e);
      }
    }

    return result;

} // End saveContact().


/**
 * This method deletes a contact from the Address Book.
 *
 * @param  inName The name of the contact to delete.
 * @param  sc     ServletContext associates with the request.
 * @return        A message saying the delete was OK.
 */
public String deleteContact(String inName, ServletContext sc) {

    log.info("\nDeleting contact:\n" + inName + "\n");

    String result = "";
```

```
// To delete a contact, we need to read in the Address Book and
// rewrite it out MINUS the contact to be deleted.  So, first thing,
// let's read it in.
ArrayList contacts = (ArrayList)retrieveContacts(sc);

// Now, let's go through and find the one to delete and do it.
for (int i = 0; i < contacts.size(); i++) {
  ContactDTO contact = (ContactDTO)contacts.get(i);
  if (contact.getName().equalsIgnoreCase(inName)) {
    contacts.remove(i);
    break;
  }
}

// Lastly, we construct a Properties object containing what is left of
// the Address Book, and write it out.
Properties props = new Properties();
int i = 1;
for (Iterator it = contacts.iterator(); it.hasNext();) {
  ContactDTO contact = (ContactDTO)it.next();
  props.setProperty("name"    + i, contact.getName());
  props.setProperty("address" + i, contact.getAddress());
  props.setProperty("note"    + i, contact.getNote());
  i++;
}
FileOutputStream fos = null;
try {
  new File(sc.getRealPath("WEB-INF") + "/" + addrBookFilename).delete();
  fos = new FileOutputStream(sc.getRealPath("WEB-INF") +
    "/" + addrBookFilename);
  props.store(fos, null);
  fos.flush();
  result = "Contact has been deleted.\n\nPlease note that if the " +
    "contact does not go away immediately, you may have to click the " +
    "Address Book link once or twice.";
} catch (IOException e) {
  log.error("Error deleting contact:");
  e.printStackTrace();
  result = "Contact could not be deleted.  Please review logs for " +
    "details.";
} finally {
  try {
    if (fos != null) {
      fos.close();
    }
  } catch (IOException e) {
    log.error("Error closing fos: " + e);
```

```
        }
      }

      return result;

  } // End deleteContact().


} // End class.
```

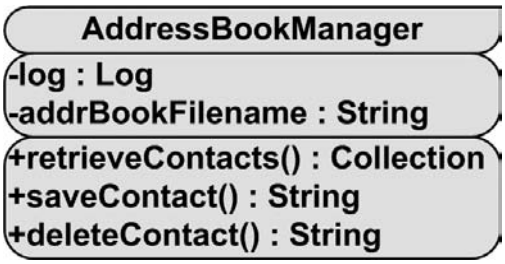


Figure 4-10. *UML diagram of the* `AddressBookManager` *class*

This is all pretty typical Java code, so I will not be going over it in detail here. There are two things I would like to point out, however. First, look at the method signatures. Do you see the `ServletContext` as an argument? In fact, look at any of the server-side class methods, except for the DTOs, and you will see this. The `ServletContext` is needed for various reasons throughout, usually to get reference to a file system location like the options file or address book file. However, look back at the client code that calls these methods . . . you won't see `ServletContext` mentioned!

As we saw in Chapter 2, DWR is able to "inject" these objects into your remote method if you include them as parameters in the client-side call to it. Using this handy mechanism here alleviates the need to write code that is coupled to DWR on the server, which is a clean way to write the code.

The second thing I want to explain is the division by 3 you will see in the `retrieveContact()` method. Recall that for each contact we are saving three pieces of information: a name, an address, and a note. Therefore, when we read in the address book, we'll have three elements in our `Properties` object for each contact. To get the total number of contacts then, we need to divide by 3.

MailSender.java

Now we come to the home stretch: the last three classes we need to look at: `MailSender`, `MailRetriever`, and `MailDeleter`. Clearly these are the classes that will do the majority of the work in InstaMail. After all, it is a mail client, and these classes clearly are working with mail messages, so let's jump right in to `MailSender`, as shown in Listing 4-5 and the UML diagram in Figure 4-11.

Listing 4-5. *The* `MailSender` *Class Code*

```
package com.apress.dwrprojects.instamail;


import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import javax.mail.Session;
import javax.mail.Message;
import java.util.Date;
import java.text.SimpleDateFormat;
import javax.mail.internet.MimeMessage;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import java.util.Properties;
import javax.servlet.ServletContext;


/**
 * This class is responsible for sending e-mails.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>.
 */
public class MailSender {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(MailSender.class);


  /**
   * This method sends a message.
   *
   * @param  inTo      The recipient of the message.
   * @param  inSubject The subject of the message.
   * @param  inText    The text of the message.
   * @return           A string indicating success or failure.
   */
  public String sendMessage(String inTo, String inSubject, String inText,
    ServletContext sc) {

    Transport          transport = null;
    FileOutputStream   fos = null;
```

```
ObjectOutputStream oos = null;
String             result = "";

try {

  // Get the options, and also get the current date/time.  We do it once
  // here so just in case we cross a second boundary while processing,
  // we know the file name and the sent time will jive.
  OptionsDTO options = new OptionsManager().retrieveOptions(sc);
  Date       d       = new Date();
  log.info("options = " + options + "\n\n");
  // Construct Properties JavaMail needs.
  Properties props = new Properties();
  props.setProperty("mail.transport.protocol", "smtp");
  props.setProperty("mail.host",               options.getSmtpServer());
  if (options.getSmtpServerRequiresLogin().equalsIgnoreCase("true")) {
    props.setProperty("mail.user",     options.getSmtpUsername());
    props.setProperty("mail.password", options.getSmtpPassword());
  }
  log.info("props = " + props + "\n\n");
  // Create a JavaMail message.
  Session session = Session.getDefaultInstance(props, null);
  log.info("session = " + session + "\n\n");
  transport = session.getTransport();
  log.info("transport = " + transport + "\n\n");
  MimeMessage message = new MimeMessage(session);
  // Populate the data for the message.
  message.addRecipient(Message.RecipientType.TO, new InternetAddress(inTo));
  message.setFrom(new InternetAddress(options.getFromAddress()));
  message.setSubject(inSubject);
  message.setContent(inText, "text/plain");
  // Send it!
  transport.connect();
  transport.sendMessage(message,
    message.getRecipients(Message.RecipientType.TO));

  // We also need to save the message.  It will be saved in WEB-INF,
  // and the file name will be the current date and time, formatted nicely,
  // with "msg_" appended on the front.  That way we can easily list them
  // all later.  Note that all we are going to do is simply create a
  // MessageDTO, and serialize it.
  MessageDTO mDTO = new MessageDTO();
  mDTO.setFrom(options.getFromAddress());
  mDTO.setTo(inTo);
  mDTO.setSent(d.toString());
  mDTO.setReceived(null);
  mDTO.setSubject(inSubject);
```

```
      mDTO.setMsgText(inText);
      mDTO.setMsgType("sent");
      String filename = new SimpleDateFormat("MM_dd_yyyy_hh_mm_ss_a").format(d);
      filename = "msg_" + filename.toLowerCase();
      mDTO.setFilename(filename);
      fos = new FileOutputStream(sc.getRealPath("/WEB-INF") + "/" + filename);
      oos = new ObjectOutputStream(fos);
      oos.writeObject(mDTO);
      oos.flush();
      fos.flush();

      result = "Message has been sent";

    } catch (Exception e) {
      e.printStackTrace();
      log.error("Error sending message");
      result = "Error sending message: " + e;
    } finally {
      try {
        if (transport != null) {
          transport.close();
        }
        if (oos != null) {
          oos.close();
        }
        if (fos != null) {
          fos.close();
        }
      } catch (Exception e) {
        log.error("Exception closing transport, oos or fos: " + e);
      }
    }

    return result;

  } // End sendMessage().


} // End class.
```

Figure 4-11. *UML diagram of the* `MailSender` *class*

`MailSender` has only a single purpose in life: to send a message via SMTP. As such, it has only a single method, `sendMessage()` (clearly I was not in a creative mode when it came to naming these three classes!). When a message is sent, `MailSender` also has to write it to disk since we want to retain all sent messages locally.

First we need to populate a `Properties` object for use with JavaMail, the standard Java API for working with e-mail. This object contains details like the SMTP server address, the username and password to use, and so forth. All of this is mandated by the JavaMail API. Once that object is ready, we open a session with the server and construct a `MimeMessage` object. This contains the text of the message, the recipient address, and the subject, among other things. Finally, the server is asked to send the message.

Once that is complete, we still have to save the local copy. To do this we populate a `MessageDTO` object. The file name we'll be saving is constructed as a string beginning with `msg_` and then containing the current date and time, down to the second. This should ensure, barring someone with a *very* quick finger or the computer having an incorrect date and time set, that each e-mail sent has a unique file name. Because all the messages begin with a known string `msg_`, we can easily list them later, as you'll see in a moment. Finally, we use plain old Java serialization to save the message. Note that error handling is pretty minimal. If an error occurs, there likely is not a whole lot we can do about it, so we simply make sure that it gets reported to the user and that the details are logged. We also ensure that the connection to the server gets closed no matter what via `finally`, since otherwise no other mail client would be able to access the mailbox. (I found this out the hard way when, during testing, I locked my primary e-mail account! It was not until I realized I needed to shut down my Tomcat instance to close the connection that I was able to access it again.)

MailRetriever.java

Now that we have seen how messages are sent, let's see how they are received. Listing 4-6 shows the `MailRetriever` class, which is where this happens, and Figure 4-12 shows the associated UML diagram.

Listing 4-6. *The* `MailRetriever` *Class Code*

```
package com.apress.dwrprojects.instamail;


import java.util.Properties;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.io.File;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.FileFilter;
import javax.servlet.ServletContext;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import javax.mail.Store;
import javax.mail.Session;
```

```
import javax.mail.Folder;
import javax.mail.Message;


/**
 * This class is responsible for getting lists of messages and individual
 * messages.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>.
 */
public class MailRetriever {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(MailRetriever.class);


  /**
   * This method retrieves the contents of the Inbox.
   *
   * @param  sc  ServletContext of the incoming request.
   * @return     A collection of MessageDTOs.
   */
  public Collection getInboxContents(ServletContext sc) {

    Collection<MessageDTO> messages = new ArrayList<MessageDTO>();
    Folder     folder   = null;
    Store      store    = null;

    try {

      // Get the fromAddress from Options.
      OptionsDTO options = new OptionsManager().retrieveOptions(sc);
      log.info("options = " + options);
      String fromAddress = options.getFromAddress();

      Properties props = new Properties();
      props.setProperty("mail.transport.protocol", "pop3");
      props.setProperty("mail.host", options.getPop3Server());
      if (options.getPop3ServerRequiresLogin().equalsIgnoreCase("true")) {
        props.setProperty("mail.user",     options.getPop3Username());
        props.setProperty("mail.password", options.getPop3Password());
      }
      log.info("props = " + props);
```

```
    Session session = Session.getDefaultInstance(new Properties());
    log.info("session = " + session);
    store = session.getStore("pop3");
    store.connect(options.getPop3Server(), options.getPop3Username(),
      options.getPop3Password());
    log.info("store = " + store);
    folder = store.getFolder("INBOX");
    folder.open(Folder.READ_ONLY);
    log.info("folder = " + folder);
    int count = folder.getMessageCount();
    for(int i = 1; i <= count; i++) {
      Message message = folder.getMessage(i);
      MessageDTO mDTO = new MessageDTO();
      // Get from address.  Note that it will have quotes around it, so
      // we'll need to remove them.
      String from = message.getFrom()[0].toString();
      from = from.replaceAll("\"", "");
      mDTO.setFrom(from);
      mDTO.setTo(fromAddress);
      mDTO.setReceived(message.getSentDate().toString());
      mDTO.setSubject(message.getSubject());
      mDTO.setSent(null);
      mDTO.setMsgID(new Integer(i).toString());
      mDTO.setMsgType("received");
      mDTO.setFilename(null);
      messages.add(mDTO);
    }

  } catch (Exception e) {
    e.printStackTrace();
    log.error("Could not retrieve Inbox contents: " + e);
  } finally {
    try {
      if (folder != null) {
        folder.close(false);
      }
      if (store != null) {
        store.close();
      }
    } catch (Exception e) {
      log.error("Error closing folder or store: " + e);
    }
  }

  return messages;
```

```
} // End getInboxContents().


/**
 * This method retrieves the contents of the Sent Messages folder.
 *
 * @param  sc  ServletContext of the incoming request.
 * @return     A collection of MessageDTOs.
 */
public Collection getSentMessagesContents(ServletContext sc) {

  Collection<MessageDTO> messages = new ArrayList<MessageDTO>();

  try {

    // First, get a list of File objects for all the messages stored
    // in WEB-INF.
    String path = sc.getRealPath("WEB-INF");
    File   dir  = new File(path);
    FileFilter fileFilter = new FileFilter() {
      public boolean accept(File file) {
        if (file.isDirectory()) {
          return false;
        }
        if (!file.getName().startsWith("msg_")) {
          return false;
        }
        return true;
      }
    };
    File[] files = dir.listFiles(fileFilter);
    if (files == null) {
      log.info("Directory not found, or is empty");
    } else {
      // Now that we know there are messages and we have a list of them,
      // go through each and reconstitute the serialized MessageDTO
      // for each and add it to the collection.
      for (int i = 0; i < files.length; i++){
        log.info("Retrieving '" + files[i] + "'...");
        FileInputStream   fis = new FileInputStream(files[i]);
        ObjectInputStream oos = new ObjectInputStream(fis);
        MessageDTO message = (MessageDTO)oos.readObject();
        oos.close();
        fis.close();
        messages.add(message);
      }
    }
```

```
  } catch (Exception e) {
    e.printStackTrace();
    log.error("Error retrieving sent messages list: " + e);
  }

  return messages;

} // End getSentMessagesContents().


/**
 * This method retrieves a single message and all the pertinent details of it.
 *
 * @param  msgType  The type of message to retrieve, either "sent" or
 *                  "retrieved".
 * @param  filename The name of the file the message is stored on disk under
 *                  if retrieving a Sent Message, null otherwise.
 * @param  msgID    The ID of the message if retrieving a message in the
 *                  Inbox, null otherwise.
 * @param  sc       ServletContext of the incoming request.
 * @return          The message requested.
 */
public MessageDTO retrieveMessage(String msgType, String filename,
  String msgID, ServletContext sc) {

  MessageDTO message = null;

  try {
    log.info("msgType = " + msgType);
    log.info("filename = " + filename);
    log.info("msgID = " + msgID);
    if (msgType.equalsIgnoreCase("sent")) {
      // Message from Sent Messages.
      String path = sc.getRealPath("WEB-INF");
      filename = path + File.separatorChar + filename;
      log.info("Retrieving '" + filename + "'...");
      File   dir  = new File(filename);
      FileInputStream   fis = new FileInputStream(filename);
      ObjectInputStream oos = new ObjectInputStream(fis);
      message = (MessageDTO)oos.readObject();
      oos.close();
      fis.close();
    }
    if (msgType.equalsIgnoreCase("received")) {
      // Message from Inbox.
      OptionsDTO options = new OptionsManager().retrieveOptions(sc);
      log.info("options = " + options);
      String fromAddress = options.getFromAddress();
```

```
        Properties props = new Properties();
        props.setProperty("mail.transport.protocol", "pop3");
        props.setProperty("mail.host", options.getPop3Server());
        if (options.getPop3ServerRequiresLogin().equalsIgnoreCase("true")) {
          props.setProperty("mail.user",     options.getPop3Username());
          props.setProperty("mail.password", options.getPop3Password());
        }
        log.info("props = " + props);
        Session session = Session.getDefaultInstance(new Properties());
        log.info("session = " + session);
        Store store = session.getStore("pop3");
        store.connect(options.getPop3Server(), options.getPop3Username(),
          options.getPop3Password());
        log.info("store = " + store);
        Folder folder = store.getFolder("INBOX");
        folder.open(Folder.READ_ONLY);
        log.info("folder = " + folder);
        int count = folder.getMessageCount();
        int i = Integer.parseInt(msgID);
        Message msg = folder.getMessage(i);
        message = new MessageDTO();
        // Get from address.  Note that it will have quotes around it, so
        // we'll need to remove them.
        String from = msg.getFrom()[0].toString();
        from = from.replaceAll("\"", "");
        message.setFrom(from);
        message.setTo(fromAddress);
        message.setReceived(msg.getSentDate().toString());
        message.setSubject(msg.getSubject());
        message.setSent(null);
        message.setMsgID(new Integer(i).toString());
        message.setMsgType("received");
        message.setFilename(null);
        message.setMsgText(msg.getContent().toString());
        folder.close(false);
        store.close();
      }
    } catch (Exception e) {
      e.printStackTrace();
      log.error("Error retrieving message: " + e);
    }

    return message;

  } // End retrieveMessage().


} // End class.
```

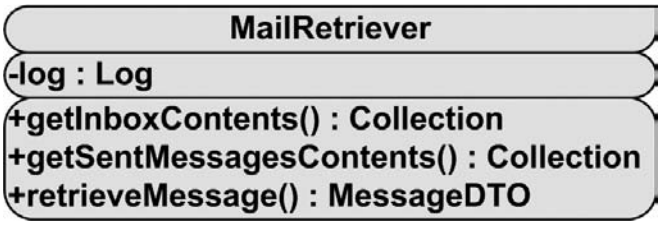


Figure 4-12. *UML diagram of the* `MailRetriever` *class*

The first method we encounter is `getInboxContents()`. Similar to the `sendMessage()` method, we begin by populating a `Properties` object. Using that object, we then get a connection to the POP3 server and ask for the contents of the `INBOX` folder. This is one of the standard folder names that all POP3 servers understand. The JavaMail API gives us access to the number of messages in the folder we are working with, which is precisely what we need. Each e-mail in a POP3 folder has a number, starting with 1, so all we need to do is loop as many times as there are messages. For each, we retrieve it, and construct a `MessageDTO` object from the pertinent details (from, subject, receive time, etc.). These objects are added to a `List`, which is returned to the caller. DWR understands Lists intrinsically, so there is no extra work to do here.

The next method we see is `getSentMessagesContents()`. This is again pretty standard Java code. We first retrieve a list of all the files in the `WEB-INF` directory beginning with `msg_`. Then for each we use the standard Java serialization mechanism to reconstitute our `MessageDTO` object. Each is again added to a `List` that is returned, and that's it! Certainly there are better ways to persist messages than object serialization, but without getting into a true persistence framework, I dare say the code is not likely to be any more concise and simple than this.

Finally, we have the `retrieveMessage()` function. This service requests to view an individual message for both the Inbox and Sent Messages views, so as expected, there is some branching logic. I thought about breaking these out into separate methods, but this way keeps the client code simpler, which to me is a more important consideration than the server-side code. And it is not as if a simple `if` branch is anything complicated.

In any case, the first branch takes us into retrieving a sent message. This is virtually identical to what we saw in the previous method, except that we are looking for a specific file name, which we are passed by the client-side code. So, we load it, reconstitute the object, and return it.

The other branch takes us into retrieving a message from the Inbox. Once again, this code is very much like that in the `getInboxContents()` method. Naturally there is no need to loop through the messages; we simply ask for the one we are interested in based on the `msgID` number passed in (which is just the index number of the message in the Inbox). Once we have it, we populate a `MessageDTO` object from it and return it to the caller.

MailDeleter.java

Only one thing is left to look at: the `MailDeleter` class, as shown in Listing 4-7 and Figure 4-13 for the UML diagram.

Listing 4-7. *The* `MailDeleter` *Class Code*

```
package com.apress.dwrprojects.instamail;


import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import javax.mail.Session;
import javax.mail.Flags;
import javax.mail.Message;
import java.util.Date;
import java.text.SimpleDateFormat;
import java.util.StringTokenizer;
import javax.mail.internet.MimeMessage;
import java.io.FileOutputStream;
import java.io.File;
import java.io.ObjectOutputStream;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.Store;
import javax.mail.Folder;
import java.util.Properties;
import javax.servlet.ServletContext;


/**
 * This class is responsible for deleting e-mails.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>.
 */
public class MailDeleter {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(MailDeleter.class);


  /**
   * This method deletes a message.
   *
   * @param  msgType   The type of message to delete, either "sent" or
   *                   "retrieved".
   * @param  filenames The names of the files the messages are stored on disk
   *                   under if deleting Sent Messages, null otherwise.  This
   *                   can be a comma-separated list of file names, or just
   *                   a single value.
```

```
 * @param  msgIDs    The IDs of the messages if deleting messages in the
 *                   Inbox, null otherwise.  This can be a comma-separated
 *                   list of IDs, or just a single value.
 * @param  sc        ServletContext of the incoming request.
 * @return           A string indicating success or failure.
 */
public String deleteMessages(String msgType, String filenames, String msgIDs,
  ServletContext sc) {

  log.info("\nAbout to delete message:\n" +
    "msgType = " + msgType + "\n" +
    "filenames = " + filenames + "\n" +
    "msgIDs = " + msgIDs + "\n");

  String result = "Message(s) deleted.";
  Store  store  = null;
  Folder folder = null;

  try {
    if (msgType.equalsIgnoreCase("sent")) {
      // Deleting from Sent Messages.
      StringTokenizer st = new StringTokenizer(filenames, ",");
      String errs = "";
      String path = sc.getRealPath("WEB-INF");
      while (st.hasMoreTokens()) {
        String fn = st.nextToken();
        fn = path + File.separatorChar + fn;
        boolean success = (new File(fn)).delete();
        if (!success) {
          if (!errs.equalsIgnoreCase("")) {
            errs += ", ";
          }
          errs += "Unable to delete '" + fn + "'";
        }
      }
      if (!errs.equalsIgnoreCase("")) {
        result = errs;
      }
    }
    if (msgType.equalsIgnoreCase("received")) {
      // Deleting from Inbox.
      OptionsDTO options = new OptionsManager().retrieveOptions(sc);
      log.info("options = " + options);
      StringTokenizer st = new StringTokenizer(msgIDs, ",");
      Properties props = new Properties();
      props.setProperty("mail.transport.protocol", "pop3");
      props.setProperty("mail.host", options.getPop3Server());
```

```
        if (options.getPop3ServerRequiresLogin().equalsIgnoreCase("true")) {
          props.setProperty("mail.user",     options.getPop3Username());
          props.setProperty("mail.password", options.getPop3Password());
        }
        log.info("props = " + props);
        Session session = Session.getDefaultInstance(new Properties());
        log.info("session = " + session);
        store = session.getStore("pop3");
        store.connect(options.getPop3Server(), options.getPop3Username(),
          options.getPop3Password());
        log.info("store = " + store);
        folder = store.getFolder("INBOX");
        folder.open(Folder.READ_WRITE);
        log.info("folder = " + folder);
        while (st.hasMoreTokens()) {
          String msgID = st.nextToken();
          int i = Integer.parseInt(msgID);
          Message message = folder.getMessage(i);
          message.setFlag(Flags.Flag.DELETED, true);
        }
      }
    } catch (Exception e) {
      log.error("Exception deleting POP3 message(s): " + e);
      e.printStackTrace();
      result = "An error occurred deleting message(s).  Please refer to " +
        "the logs for details.";
    } finally {
      try {
        if (folder != null) {
          folder.close(true);
        }
        if (store != null) {
          store.close();
        }
      } catch (Exception e) {
        log.error("Error closing folder or store: " + e);
      }
    }

    return result;

  } // End deleteMessages().


} // End class.
```

Figure 4-13. *UML diagram of the* `MailDeleter` *class*

Just like the `MailSender` class, there is only a single method here to look at: `deleteMessages()`. Recall from our earlier look at the client-side code that this method expects to receive a comma-separated list of items to delete, either file names when dealing with sent messages, or message IDs when dealing with received messages. So, after we branch based on whether we are deleting sent messages or received messages, as you might expect, the first thing we need to do is tokenize the string. When that is done, we enter a loop until we have exhausted all the tokens. For each, in the case of sent messages, we get a reference to the file and delete it using the standard `File.delete()` method. If any errors are encountered, we record them in a string to be returned to the caller for display, but we keep going until no tokens are left.

For received messages, it is only slightly more difficult. One last time we see the `Properties` object being populated. We also see a session established with the POP3 server. We then loop, and for each token we retrieve the message from the server. (Remember that the token is the message number.) The way you delete a message with the JavaMail API is to retrieve it, and then set a flag marking it for deletion. The message will still not actually be deleted until you close the folder, and then only if you pass `true` to the `close()` method.

The same kind of minimalist error handling that we have seen before is again used. The outcome of the operation is returned to the caller for display.

And that wraps it up! We have just explored every part of InstaMail, and have experienced our first full DWR-based application. I hope our initial foray into DWR has convinced you that DWR really made it very nice, neat, and compact in terms of code size for the power it provides.

Suggested Exercises

I have purposely left open a number of items that I encourage you to pursue to make InstaMail even better. I believe these items will give you good experience in using DWR. None of them should prove especially difficult, either.

- Allow for multiple e-mail accounts. Tangential to this is the ability for multiple users to log in and use a specific account.
- Allow for file attachments. I highly suggest looking at the Commons FileUpload component (`http://jakarta.apache.org/commons/fileupload`) for this. It will save you a great deal of time and effort and make this enhancement almost child's play!
- Allow for "rich" e-mail content. Spend some time Googling for rich edit components to integrate. A number of them are available, and many of them are Ajax-enabled already.
- Allow for custom folders. It would be nice if you could create a `Storage` folder, for example, and move any e-mail you wanted into it.

- Use a database to store everything instead of the file system. I opted to use the file system in all applications in this book because I did not want you to incur the extra burden of having to set up and configure a database. That being said, many of the applications very much fit in the database mold, and InstaMail is a prime example. This would clearly make the application more scalable and robust and would allow for further functionality.
- Add search capabilities. It should be quick and easy to find any message.
- Allow for paging of folder views. Notice how the Inbox and Sent Message views always show all their contents? Paging through them, say 25 messages at a time, would be very helpful when viewing large numbers of messages.

All of these should be pretty fun enhancements to make, and I do hope you will undertake them.

Summary

In this chapter, we built a webmail client application utilizing Ajax techniques courtesy of the DWR library, our first such application in this book. We discovered how DWR allows us to essentially treat Java classes on the server as client-side components, calling methods on them, accessing properties, and so on. We saw how what is truly a small amount of code, with the right libraries, can yield a rather useful application. In addition, we explored just a few more advanced client-side presentation techniques using DOM scripting and CSS.

In the next chapter, we'll get our second taste of DWR in a slightly more complex application, namely a wiki.

CHAPTER 5

Share Your Knowledge: DWiki, the DWR-Based Wiki

I remember a *Dilbert*[1] comic strip some time ago that I'll try to recap for you here. Since it's from memory, it may not be completely accurate, but hopefully it's close enough to do the trick.

It involved a consultant coming in trying to sell the pointy-haired boss some online collaboration tools. The boss, in an unusually insightful moment, says to the consultant, "Won't these tools just help my ignorant employees to more effectively share their ignorance?" At that point, the consultant starts frantically trying to push the boss to sign the contract!

What's the point you ask? Other than the fact that *Dilbert* is less a comic strip and more a documentary of the American corporate IT department (well, corporate America in general really), we're going to be building an online collaboration tool in this chapter! Now, whether you use it to share ignorance or useful knowledge is, we shall simply say, beyond the scope of this book.

In this chapter, we'll build a wiki, which is one of the more useful products of the whole Web 2.0 movement. We'll of course use DWR to give it that Ajax flavor it so desperately needs, and in the process we'll get ourselves a handy little application that can be used by you and yours to share your knowledge (or ignorance, depending on whether your own pointy-haired boss is right or not!). Without further delay, let's get to work (although, if you'd like to take a few minutes now to go browse some *Dilbert* archive somewhere, I wouldn't be upset!).

Application Requirements and Goals

So, what exactly is a wiki anyway? Simply put, it's a web application that allows users to create a web site collaboratively. In other words, a user can create a page, often called an *article*, and can then create other articles and link them. These linkages are generally created automatically without having to write actual HTML, which in general isn't required for working on a wiki. In most cases, only registered users are allowed to modify and create articles, allowing

1. If you are unfamiliar with *Dilbert*, I highly suggest you kick off that rock you've obviously been living under and check it out: `www.dilbert.com`. *Dilbert* is an American comic strip with a collection of hysterical characters all inhabiting a corporate IT department (well, it **seems** to be an IT department, although I'm not sure that's ever stated one way or another). Either way, you yourself likely work in IT if you're reading this book, and I suspect you'll identify and get a great deal of enjoyment out of creator Scott Adams' comic creation.

for some degree of accountability. The wiki has to deal with concurrency issues where two users want to edit the same article at the same time (this is usually accomplished by one user "locking" the article for editing).

A number of other features are often present in a wiki, and we're going to build in a couple of them here. Let's go ahead and list out the features and functionality DWiki should provide:

- As mentioned previously, only registered users will be able to add or modify articles (or comment on them).
- Speaking of commenting on articles, we'll allow users to discuss a given article without actually editing it. All users should be able to read the comments, but only registered users can leave them.
- We'll implement security using container-managed security, locking down the applicable remoted methods via DWR's integration with J2EE security.
- When edits are made to an article, a history is kept so that people can see who changed what. For the sake of simplicity, we'll simply record the text of the article before and after the edit (not very sophisticated, but easy to implement).
- We'll have a help page so people can understand how to use DWiki.
- Although HTML isn't required to write articles, let's allow it to be used so that rich content can be included in articles without having to create a whole wiki language ourselves.
- We want DWiki to be as flexible as possible, so we'll want to have as many of the screens the user will see to be as editable as possible. Let's make as many of them as we can really just be articles, like any other. There will be a couple of exceptions though, and to deal with them we'll use a flexible templating system called FreeMarker, and we'll create a generic DWR extension to use it that you can reuse later.
- Any word that begins with a capital letter will be interpreted as a link to an article. Clicking such a word will lead to the article or to a placeholder where the user can create the article if it doesn't exist. We'll also need a way to override this linkage because clearly not **all** words beginning with a capital letter should be article links (this is again not a very sophisticated algorithm, but it gets the job done).
- We'll use a database for storing the wiki content, Apache's Derby to be specific, and let's try and avoid writing as much low-level JDBC code as we can using the Spring library's JDBC support.
- DWiki will support the concept of "locking" an article for editing, and the lock will last for some configurable period of time.

Naturally, DWR will be used to connect the user interface with the back end, which as you'll see allows the amount of code we have to write to be not nearly as large as it otherwise could have been.

Before we can really get into dissecting the code of the application, we should discuss a few key pieces of technology that we'll be making use of, namely the FreeMarker templating library, Apache's Derby database, and the Spring library's JDBC support.

FreeMarker

For no real reason other than I like being a mite unpredictable from time to time, let's start with FreeMarker. And because I also like being lazy whenever humanly possible (I worship at the altar of Homer Simpson for sure), I'll explain what FreeMarker is by liberally interpreting the description on the FreeMarker home page (if you take "liberally interpret" to mean quote directly, that is!).

> *FreeMarker is a "template engine"; a generic tool to generate text output (anything from HTML to autogenerated source code) based on templates. It's a Java package, a class library for Java programmers. It's not an application for end-users in itself, but something that programmers can embed into their products.*
>
> *FreeMarker is designed to be practical for the generation of* HTML Web pages, *particularly by servlet-based applications following the* MVC *(Model View Controller) pattern. The idea behind using the MVC pattern for dynamic Web pages is that you separate the designers (HTML authors) from the programmers. Everybody works on what they are good at. Designers can change the appearance of a page without programmers having to change or recompile code, because the application logic (Java programs) and page design (FreeMarker templates) are separated. Templates do not become polluted with complex program fragments. This separation is useful even for projects where the programmer and the HMTL page author is the same person, since it helps to keep the application clear and easily maintainable.*
>
> *Although FreeMarker has some programming capabilities, it is* not *a full-blown programming language like PHP. Instead, Java programs prepare the data to be displayed (like issue SQL queries), and FreeMarker just generates textual pages that display the prepared data using templates.*

WHAT IS A TEMPLATE ENGINE ANYWAY?

In short, a *template engine*, or *template processor* as it's sometimes called, is a piece of software or a component that can be used to build software, that takes something called a *template*, which is akin to a form letter with blanks to be filled in, and combines it with a *data model*, that is, a collection of data in an organized format, and produces some sort of document output, be it plain text, a PDF, an Excel spreadsheet, or what have you. FreeMarker is, of course, one such template, and so it is something else you're already familiar with: JavaServer Pages! Although some people don't realize JSP is a templating technology, it in fact is.

Yep, that certainly says it all! Well, **almost** all. I imagine you'd like to see what FreeMarker actually looks like in practice, huh? Your wish is my command: Listing 5-1 shows a very simple FreeMarker template.

Listing 5-1. *The Hello World of FreeMarker Templates*

```
Hello, ${name}, from a Freemarker template!
These are a few of my favorite things:
<#list favoriteThings as thing>
  ${thing.name}
</#list>
```

Now, as you'll note in the last paragraph of the quoted FreeMarker description, ". . . Java programs prepare the data to be displayed. . . ." In practice, this means you construct a `Map` and populate it with data. The data in the `Map` is used to replace the tokens, such as `${name}` in the template. In Listing 5-2, you can see a simple example of creating this data `Map` for use with the template in Listing 5-1.

Listing 5-2. *A Very Simple Example of Using FreeMarker*

```
import freemarker.cache.ClassTemplateLoader;
import freemarker.template.Configuration;
import freemarker.template.DefaultObjectWrapper;
import freemarker.template.Template;
import java.io.StringWriter;
import java.io.Writer;
import java.util.ArrayList;
import java.util.HashMap;

public class FreemarkerTest {

  public static void main(String[] args) {

    try {

      Configuration fmConfig = new Configuration();
      fmConfig.setObjectWrapper(new DefaultObjectWrapper());
      fmConfig.setTemplateLoader(new ClassTemplateLoader(
        new FreemarkerTest().getClass(), "/"));

      HashMap data = new HashMap();
      data.put("name", "MY FAVORITE READER");
      ArrayList favoriteThings = new ArrayList();
      HashMap ft1 = new HashMap();
      ft1.put("name", "Soda");
      favoriteThings.add(ft1);
      HashMap ft2 = new HashMap();
      ft2.put("name", "Pizza");
      favoriteThings.add(ft2);
      HashMap ft3 = new HashMap();
      ft3.put("name", "Jericho");
      favoriteThings.add(ft3);
      data.put("favoriteThings", favoriteThings);
```

```
      Template template = null;
      template = fmConfig.getTemplate("hello.ftl");
      Writer out = new StringWriter();
      template.process(data, out);
      out.flush();
      System.out.println(out.toString());

    } catch (Exception e) {
      e.printStackTrace();
    }

  }

}
```

This is a 100 percent working example that you can find in the source download bundle for this book. Compile and run it, and you'll see the output shown in Figure 5-1 (you can also see the command line to execute to both compile and execute).

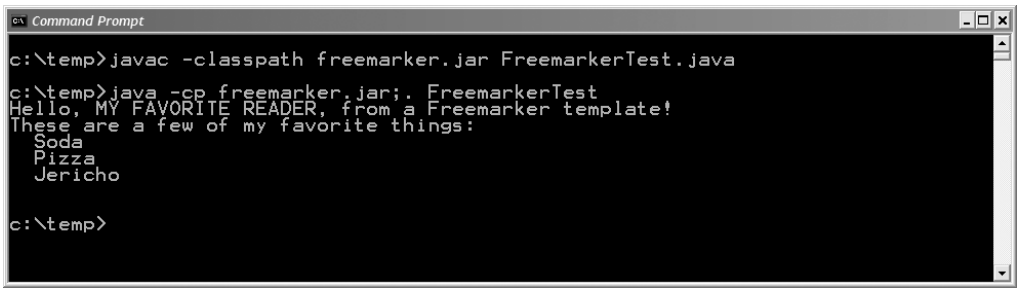


Figure 5-1. *FreeMarker is alive!*

The code, I think, is fairly self-explanatory. After some initial FreeMarker setup, which basically amounts to creating a `Configuration` object and telling it how to view the data model (that's the object wrapper stuff, but that's a more advanced topic that we'll skip for the moment) and telling FreeMarker how to find templates (in this case, it's a path relative to the class using FreeMarker, so the template has to be in the default package alongside the class). Once that's done, you populate a `Map` in typical fashion and finally instantiate the template (more specifically, you tell the `Configuration` object which template you want and let it instantiate it) and have it process the template using the data `Map`. The output is written to the console, and that's that. Very easy!

FreeMarker offers a good selection of programming-like constructs that you can use such as loops, branching, and variables, but as the FreeMarker description itself states, it is in no way, shape, or form meant as a full-on programming language. Keeping the templates as simple and vanilla as you can is probably advisable. The templates in the DWiki application aren't much more complex than this first example, so we won't be going into much more detail on using FreeMarker than what you've seen here. But I think you'll agree, FreeMarker is a pretty handy little tool to have around!

We're only touching on FreeMarker enough in this book to accomplish our goals, but it can do quite a bit more than you'll see here. FreeMarker is used in many well-known applications and frameworks, Struts 2 being one prominent example, so it has a very good pedigree. I suggest taking some time to check out the FreeMarker web site at `http://freemarker.sourceforge.net`. One nice thing about FreeMarker is that its documentation is quite good, and the FreeMarker team takes the time to walk you through some good examples. I think you'll find them very informative. Speaking from personal experience, I only looked at FreeMarker a few months ago, but since then I've found myself using it more and more, fitting it effortlessly into situations that I used to write code for. It's one of the more useful libraries I've come across in recent years, so do take the time to check it out—I think you'll find it just as useful as I have.

Apache Derby

The next piece of the puzzle is how and where to store the content that users create on DWiki. We have, of course, some choices here. We could store content on the file system, which was actually what I originally had planned. That has a host of problems associated with it though, most importantly that in some environments you won't be able to write to the file system from a webapp, and since I wanted DWiki to be at least somewhat useful for you, dear reader, I couldn't assume you'd want to run it in an environment that wouldn't allow it.

Another approach is to use a database, and that's exactly what I decided to do, and that's exactly where Derby comes in.

Derby began its life as a product called JBMS of a company called Cloudscape, Inc. Its initial release was in 1997. A short time later, in 1999 to be precise, Informix acquired Cloudscape, and took JBMS as part of that acquisition. The corporate acquisition train kept steaming down the track, and in 2001, IBM purchased assets of Informix, including JBMS. IBM subsequently renamed it IBM Cloudscape and released it to the masses. Finally, in August 2004, IBM donated Cloudscape to the Apache Foundation, where it was renamed Apache Derby, and the rest is history (err, so was **that**, of course!).

Anyway, that's the history lesson for today. Now, what exactly is Derby? That one's easy: Derby is a relational database that is entirely Java-based. Derby has a relatively small footprint, and more importantly for our purposes here today, is embeddable, meaning you can make it a component of an application very easily, owing to the fact that it has a built-in JDBC driver. Derby can also be used in a more traditional client-server mode, if that's something you need.

Best of all, Derby is rather easy to install and use!

To "install" derby, all you need to do is add the `derby.jar` file to your project, and you're off to the races! Beyond that, using Derby is essentially the same as using any other RDBMS you are familiar with. So, to get started, you can load the Derby JDBC driver like so:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
```

So long as you have the Derby JAR in your classpath, that'll work just peachy. Once the driver is loaded, you can proceed to get a connection to the database:

```
Properties props = new Properties();
props.put("user", "user1");
props.put("password", "user1");
Connection conn =
  DriverManager.getConnection("jdbc:derby:testDB;create=true", props);
```

One of the neat things Derby can do for you is that if a database doesn't exist yet, it can be created on the fly. That's what the `create=true` part of the connection URI does for us. This means you can get up and running, developing an application, very quickly and easily.

Once you have the connection, you proceed to do standard JDBC programming, no different from any other JDBC programming. You can, of course, use something like Hibernate or iBATIS if you prefer (assuming they support Derby, which Hibernate definitely does and iBATIS appears to do). You can also use the Spring Framework's JDBC support, which is exactly what we'll be doing here in DWiki, and what we'll be talking about next.

This is clearly just a brief introduction to what Derby is. We'll see more usage of it as we go through the DWiki code, but Derby offers a lot more than you'll see in this chapter, so exploring it on your own is definitely something I recommend. You'll find that, in particular, the embedded mode is an extremely handy way of developing an application. For instance, where I work we use Oracle for all our database needs. However, we have lots of procedural red tape to get through to create or later modify a database. So, by using Derby in embedded mode, I'm able to develop the basic database schema, and get the application basically working, and only then deal with setting up Oracle and tying into that. Aside from that, being embeddable means that if you want to distribute an application that uses a database, say like, oh, I don't know, a wiki application in a book, you can do so, and then the person using the application (or reading the book) doesn't have to deal with a database setup unless he or she wants to. So, I guess you'll want a URL, huh? Here you go: `http://db.apache.org/derby`.

Spring JDBC

Unless you have been living under the proverbial rock for the past two years or so, you have almost certainly heard of the Spring Framework. You most likely heard about it first in the context of dependency injection, or IoC (Inversion of Control). This is probably what it is most well known for. However, Spring is much more than that, as we're about to see, at least to some degree.

Spring is what is termed a *full-stack* framework, meaning it pretty well covers all the bases a J2EE developer might need. It takes a layered approach, meaning that each "module" of functionality can be, more or less, used independently, and you can add only the functionality you need. Spring runs the gamut from dependency injection as mentioned, to web Model-View-Controller (MVC), to various general-purpose utility functions such as string manipulations and such, Object-Relational Mapping (ORM), and JDBC.

Speaking of JDBC, that's in fact the unit of functionality DWiki needs. The Spring JDBC package includes classes that make working with JDBC easier and, more importantly, less error prone. One of the banes of JDBC programming is that developers are sometimes forgetful beasts and neglect to do things like release database connections when they have finished with them. No amount of hand-slapping with a ruler will solve the problem, unfortunately. This leads to resource leaks and eventually a crashed application, or worse yet, an entire server. With Spring JDBC, these types of mistakes are virtually impossible.

With Spring JDBC, database access basically boils down to two steps. First, get a data source to work with, and second, execute the pertinent SQL. For step one, the following code is used:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName(Config.getDatabaseDriver());
dataSource.setUrl(Config.getDatabaseURI());
dataSource.setUsername(Config.getDatabaseUsername());
dataSource.setPassword(Config.getDatabasePassword());
```

The `Config` class, as we'll see later, contains a number of configuration parameters used in DWiki, including the details of connecting to the database, such as the driver to use, the URL, the username, and the password. Once we have a data source, step two is accomplished something like this:

```
JdbcTemplate jt = new JdbcTemplate(dataSource);
List rows = jt.queryForList("SELECT * FROM myTable");
```

The `JdbcTemplate` class is our gateway into the world of Spring JDBC. As you can see, it provides a `queryForList()` method, which returns us a `List` representing the result set. No more messing around with `ResultSet` objects! Other methods are provided, such as `queryForMap()`, which is used to retrieve a single record from the database and get a `Map` in return, making it very simple to access the fields of the record. Note that I am not showing any connection-handling code, no cleanup code, things of that nature. That is because the data source and template handle all that for us. No worries about closing the connection or dealing with statements and that sort of typical JDBC work.

WHY HAS SPRING BECOME SO POPULAR SO FAST?

The rapid rise of Spring is largely due to the core beliefs that its original creator, Rod Johnson, founded the project on. His belief is that programming in the J2EE world is far too complex and can be simplified by working "around" the specs. This sounds a bit controversial (although it happens to be a nearly verbatim quote from Rod himself), but I think what he really meant was that you could build something on top of the specs that actually made them simpler. A prime example is Enterprise JavaBeans, which until the advent of EJB 3.0, was very complex and difficult for new developers to get into. It was just too complex for most people. By embracing the idea of POJOs that could be injected into classes that needed them, the Spring Framework provided most of the promised benefits of EJBs without the difficulty involved in writing them. This was the original reason Spring gained popularity.

Interestingly, some now claim that Spring itself has gotten at least as bloated and complex as the specs it sought to simplify in the first place! I'll leave that particular judgment to you, learned reader!

One last thing I'd like to mention about Spring JDBC is that it wraps `SQLExceptions` in custom exception classes so that your code can be abstracted away from the standard JDBC classes entirely. For instance, if you have constraints on a database table to stop duplicate records, the `JdbcTemplate` class will throw a `DataIntegrityViolationException` object, which you can catch and handle appropriately. The exception hierarchy offered by Spring JDBC is more intuitive and frankly useful than the JDBC exceptions, so using them is definitely a Good Thing™.

Spring JDBC frees us from all the nitty-gritty details of JDBC programming and makes it safer in the process. If you don't want (or need) to move to an ORM tool like the popular Hibernate, or EJB3, or something along those lines, I very much recommend Spring JDBC instead of straight JDBC programming. It's worth its weight in gold, not to mention saved time debugging!

The JDBC package is just one small part of Spring. A great deal more information is available to help you develop your applications better and faster. Take some time to check out what Spring has to offer; I think you will love what you find: `www.springframework.org`.

Dissecting DWiki

Alright, now it's time to get into the DWiki application itself, and we'll begin that exploration by taking a look at the directory structure and files that make up the application. In Figure 5-2, you can see that directory structure. I've left the `img` directory unexpanded because it's just a bunch of images that are used to build up the UI, and frankly, expanding it would have made the screen capture a little more difficult!

In the root directory, you find three files: `index.jsp`, `login.jsp`, and `loginOk.jsp`. The first is the main markup for the application, and the latter two are, as you'd expect, used during the user login process.

Going through the directories top to bottom, in the `css` directory we find a single file, `styles.css`, which is the style sheet for the application. Skipping over the `img` directory, we next find the `js` directory, which contains two files, `DWikiClass.js` and `RolloversClass.js`. The first is the main JavaScript that makes up the application, and the latter contains code for doing image rollovers for menu items.

After that comes the `templates` directory, which contains all the FreeMarker templates the application uses. I won't enumerate them here because we'll go over them in detail a little later.

After the `templates` directory is the standard `WEB-INF` directory, and in it you find two files, `web.xml`, which is of course the standard Java webapp descriptor, and `dwr.xml`, the DWR configuration file from DWiki.

Figure 5-2. *The directory structure of the application*

Within the `WEB-INF` directory we find the `classes` directory. Aside from being where the compiled DWiki classes go, it also contains three files. First is `dwiki.propeties`, which is the configuration file for the application. Next are two files that configure Jakarta Commons Logging, namely `commons-logging.properties` and `simplelog.properties`. The first configures the logging package itself, and the latter configures the `SimpleLog` logger, which just writes log messages out to sysout. Configuring logging is a whole topic unto itself, so I haven't devoted a section to it, but for the sake of not leaving anything unexplained, I'll quickly describe it here. If you look in `commons-logging.properties`, you'll see that it specifies the `SimpleLog` logger as the default logger. Then, in `simplelog.properties`, it defines that the default level for the `SimpleLog` logger is error, and then that any classes in the `com.apress.dwrprojects.dwiki` package are logged at debug level. This means that we'll see all but trace messages from the classes that make up DWiki, but only messages of error or fatal from any other classes (such as Tomcat). This is an appropriate level of logging by default when you're trying to develop an application like this.

After the `classes` directory comes the `lib` directory, which of course contains the entire collection of library JARs we need for this application. I've left this branch unexpanded to save some space, so I'll just quickly go over what's found there. The `commons-logging.jar` is of course Jakarta Commons Logging. The `commons-lang.jar` is Jakarta Commons Lang, which is a

collection of very useful "language extensions," so to speak, to Java itself. The `derby.jar` is the Apache Derby database JAR. I'll give you just one guess what `dwr.jar` is! The `freemarker.jar` is the FreeMarker templating engine. After that is a collection of four JARs named `spring*.jar`, where the asterisk is `beans`, `core`, `jdbc`, and `tx`. This is the minimum set of JARs from the Spring Framework required to use the JDBC package.

Next up is the `src` directory. In it you'll find the `build.xml` file, which is the Ant build script for the application. You'll also find a `build_libs` directory, which contains JARs needed during compilation but not execution (they are supplied by the servlet container at runtime). One is the servlet API JAR, the other is the JSP API JAR.

Lastly, starting in the `com` directory is the source tree and files for DWiki. Naturally, we'll be looking at each of these shortly, so I won't list them out here.

And with that, we can begin looking at the actual pieces that make up the application in detail, starting with the configuration files involved.

Because this chapter turned out to be rather large, in the interest of saving some space in print I have removed comments from the code listings throughout (any time a comment was its own line specifically) and also removed some line breaks. Rest assured, though, that when you look at the actual code, there are comments aplenty and spacing to help organize things logically.

Configuration Files

There are a grand total of four configuration files involved in configuring DWiki, and none of them are particularly complex. Still, I'd be remiss if I didn't explain each one, so off we go, as Piers Morgan[2] would say.

Container-Managed Security Configuration

In this application, we have some security requirements. Namely, a user who is not registered with the wiki cannot create or modify articles, nor can they leave comments on articles (although they can read existing comments). To accomplish this, I decided to use container-managed, or J2EE, security.

Doing so is very simple with DWR, but it all starts with configuring the container, Tomcat in this case (assuming you're running in the recommended environment). This configuration really amounts to not much more than listing out the users who will have access to the protected resources. Recall that with J2EE security, you are basically setting constraints on resources that would otherwise be served freely. In our case, those resources are methods of classes that we are remoting via DWR.

The way this container-managed security works is that you first create a role. This theoretically maps to a particular type of user; think of something like regular users vs. administrative users, for example. You then list a given user and tell that user what role or collection

---

2. Piers Morgan is one of three judges on the NBC show *America's Got Talent*. Many people say he's the Simon Cowell rip-off judge (Simon being the acerbic, yet seemingly always right, judge on *American Idol*). Yeah, OK, I admit, this particular pop-culture reference is a bit of a stretch!

of rules he or she belongs to. When you actually constrain a resource, say, for instance, you only want administrative users to be able to access any URL beginning with `/admin`, you tell the container that only users who are in a given role can access those URLs. You do **not** list individual users, you list roles.

So, it all begins with listing the roles and users. Each container is different, but in Tomcat this is accomplished very easily by adding entries to the file `tomcat-users.xml`, found in the `${TOMCAT}/conf` directory. In Listing 5-3, you can see this file as it is for DWiki.

Listing 5-3. `tomcat-users.xml` *Configuration File*

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="dwiki_user"/>
  <user username="dwiki" password="dwiki" roles="dwiki_user"/>
</tomcat-users>
```

In this case, because we're really just talking about whether a given user is registered or not, all we really need is a single role, here called `dwiki_user`. Then, we have a single user configured, `dwiki`, and you can see the password for that user specified here. You can also see the `roles` attribute of the `<user>` element, which defines what role or roles the user belongs to. This can be a comma-separated list of role names, which would match the `rolename` attribute of a `<role>` element (of which there can be many). That's all there is to it! Tomcat is now aware of the users and roles for this application.

Now, actually making use of this configuration is done at the application level, inside the `web.xml` file, so let's go there next and see how it all fits together.

Container-managed (J2EE) security is a larger topic than you can tell from this explanation, although frankly not all that much more, as it's basically a pretty simple mechanism. It can get more interesting, however, because a container/application server is free to offer whatever value-adds on top of the basic mechanism it wishes. Things like hooking into an LDAP directory for your user store is one popular example. If you'd like to dive into it a bit more, I suggest some light reading here: `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Security.html`, `www.ibm.com/developerworks/websphere/techjournal/0303_barcia/barcia.html`, and `http://tomcat.apache.org/tomcat-6.0-doc/realm-howto.html`. Keep in mind that each application server will be a little different, so you should seek that information out using your favorite search engine.

---

web.xml

With users and roles set up, the last step is to configure the application itself to constrain those resources we want to protect. Remember always that the job of a server is to **serve**, and therefore, a given resource is unprotected (unconstrained in J2EE security parlance) unless configured otherwise. To do so simply requires some entries in the standard `web.xml` file, as shown in Listing 5-4.

Listing 5-4. `web.xml` *for DWiki*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app id="dwiki" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>DWiki</display-name>

  <listener>
    <listener-class>
      com.apress.dwrprojects.dwiki.DWikiContextListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/login.jsp?login_bad=true</form-error-page>
    </form-login-config>
  </login-config>
  <security-constraint>
    <web-resource-collection>
      <url-pattern>/loginOk.jsp</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
```

```
      <role-name>dwiki_user</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <role-name>dwiki_user</role-name>
  </security-role>

  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

The part we're most interested in is, of course, the section under `<servlet-mapping>`. Let's come back to the `<login-config>` element in a moment and instead start with the `<security-constraint>` portion. This defines a resource that is going to be constrained. Here, we're saying that the `loginOk.jsp` specifically is to be protected. Further, we're saying that the `GET` method only is protected. This isn't the most solid security possible obviously, but for our purposes it's sufficient (we're not so much trying to protect resources as we are just developing a login mechanism for known users). Within the `<security-constraint>` we also have an `<auth-constraint>`. This says that this constrained resource is only accessible by users in the `dwiki_user` role.

Following the `<security-constraint>` is a `<security-role>` element. This is needed to enable a given role, as defined in `tomcat-users.xml`, to be used in this webapp. We only have one role, so we only need one `<security-role>` element, but we could, of course, have as many as we need if there were more.

Now, let's go back to that `<login-config>` element. What happens when a client requests a constrained resource? It wouldn't be great if it simply denied the request, would it? No, something else needs to happen, and it's the `<login-config>` element that tells the container what to do in those situations. More specifically, the `<auth-method>` element defines what to do. There are a couple of options here, including `BASIC`, which causes the browser to pop up a dialog box asking for username and password (this is browser-dependent). You can also use `CLIENT_CERT`, which requires the client to send a digital signature that the server recognizes. There is also `DIGEST`, which works like `BASIC`, but which requires some additional setup on the server to support MD5 encryption. Lastly, and probably the most used option, is what we have here, `FORM`.

The `FORM` auth method causes the container to redirect to some login page, which you configure with the `<form-login-page>` element, and which is configured as `login.jsp` in this application, but there is, of course, no rule to what JSP you have to use (it in fact doesn't have to be a JSP, it could be a plain HTML page, or even something like a Struts Action, if you were using Struts and configured things properly).

ALTERNATIVE TO J2EE SECURITY

The container-managed security discussed here is not the only game in town. Another viable alternative is Acegi, a popular security framework. It provides quite a few capabilities that J2EE security does not, and is by most accounts a much more robust infrastructure. Best yet, it is possible to integrate Acegi into DWR (although it's not an officially supported integration as near as I can tell). The DWR documentation provides a link to an article detailing how to do it, should you be interested (this topic is not covered here).

So, here's the sequence of events: the client requests a constrained resource, `loginOk.jsp` in our case. The container, seeing that there is a constraint on that resource, redirects the client to the configured login page, `login.jsp` in this case. On that page, as you'll see shortly, is a special (sort of!) form that the user uses to enter his or her name and password. This is submitted to a special (again, sort of!) servlet that validates the information using the configuration in `tomcat-users.xml`. Assuming the information is valid, the container then returns the resource that was originally requested, `loginOk.jsp`, and the user is then effectively logged in. If the information is not valid, the container redirects to the page specified by the `<form-error-page>` element, which in our case is simply the `login.jsp` again, but where a message will be displayed in red saying the information was invalid (note the query string on the value of the `<form-error-page>`—that is a flag that tells the JSP to display the message).

Now all the pieces of the puzzle should fit into place in your mind. All except for one, that is: this all explains how logins occur, which is something the application needs, of course, but how does it constrain the DWR-remotable classes and methods? The answer is to be found in the `dwr.xml` file, and that's where we're headed next.

dwr.xml

We've already seen the DWR configuration file a few times now, so most of this should be old hat to you by now. In Listing 5-5, you can see the configuration file.

Listing 5-5. *The DWR Configuration File,* `dwr.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>

    <convert converter="bean" match="com.apress.dwrprojects.dwiki.Article" />
    <convert converter="bean"
      match="com.apress.dwrprojects.dwiki.ArticleComment" />

    <create creator="new" javascript="ArticleDAO">
      <param name="class" value="com.apress.dwrprojects.dwiki.ArticleDAO" />
      <auth method="addArticle" role="dwiki_user" />
```

```
      <auth method="updateArticle" role="dwiki_user" />
      <auth method="addComment" role="dwiki_user" />
    </create>

  </allow>
</dwr>
```

We have two classes, `Article` and `ArticleComment`, which we'll be passing back and forth between client and server, so we need to specify converters for those, and here the basic bean converter does the trick just fine.

After that, we specify that the `ArticleDAO` class is to be remotable, and in fact that's the only class we need to remote for this application.

Now it gets a little new. This is our first encounter with the `<auth>` elements, but it couldn't be simpler. For each method you wish to constrain, you create an `<auth>` element, and in it you list the name of the method to constrain, and the role(s) that is allowed to execute it, and that's it. Any method not listed will be accessible, which is exactly what we want because only these three require a user to be registered.

Under the covers, DWR will use the `request.isUserInRole()` function to determine whether a given method invocation should be allowed or not. Once users go through the login process outlined in the last section, they get a cookie set that ties them to a table in the container that indicates what role they logged in under, and that's the information returned by `request.isUserInRole()`.

I told you DWR makes this all too easy. See, I wasn't joshin'!

dwiki.properties

The last of the configuration files is `dwiki.properties`, which is a simple Java properties file that gives DWiki some information it needs (see Listing 5-6).

Listing 5-6. *The DWiki-Specific Configuration File,* `dwiki.properties`

```
databaseDriver=org.apache.derby.jdbc.EmbeddedDriver
databaseURI=jdbc:derby:dwiki;create=true
databaseUsername=
databasePassword=
editLockTime=60
```

In Listing 5-6, you can see this information amounts to a grand total of five properties, and I'd be willing to bet you can reason out what each one is on your own, but since I get paid by the page, I'll go ahead and list them out for you anyway in Table 5-1.

Table 5-1. *The DWiki Configuration Properties*

Property Name	Description
`databaseDriver`	This is the fully-qualified class name of the JDBC driver to use to connect to the database.
`databaseURI`	This is the URI (connection string) to use to connect to the database.
`databaseUsername`	If a user is required to access the database (it isn't required for our Derby-based database by default), this is that value.

Property Name	Description
`databasePassword`	If a user is required to access the database, this is the password for the user.
`editLockTime`	When a user is editing an article, that article is locked for a duration of time so that no other user can edit it concurrently. This value defines that duration of time, in seconds.

With the database connection information externalized like this, it should be a fairly simple matter to switch to a different RDBMS if you wish. The other thing that may have to change is the code that builds the tables in the database, and we'll see that soon.

The Client-Side Code

Now that we've gone over all the configuration files, we can dive headlong into some actual code, and we'll begin with the client-side code. Before we do that though, let's take our first look at the DWiki application itself, as shown in Figure 5-3. Note that this is not precisely what you'd see after logging in. In point of fact, the very first time you access DWiki, the "FrontPage" article won't yet exist, so in fact this figure shows what it looks like only **after** you've created that article.

DWIKI - A DWR-Based Wiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/dwiki/
Google
DWIKI - A DWR-Based Wiki
DWIKI
A DWR-Based Wiki
dwiki | Login
Article Edit History Comments
Front Page
All Articles Index
Search
Help
Welcome to DWiki!
This is the front page. It can be edited, just like any other article.
I hope you enjoy using DWiki. Thanks for visiting, take care!
Last edited Wed Jul 18 2007 01:01:45 GMT-0400 (Eastern Daylight Time) by dwiki
Powered By DWR
0.094 s
169.969 s
115.44 KB
27 req
127.0.0.1
Apache-Coyote/1.1
6.800000190734863 MB / 27 MB
9.800000190734863 MB / 24.399999618530273 MB

Figure 5-3. *The DWiki front page, what the user sees initially*

As you can see, the screen is broken into a number of sections. First is the top area, where the title is on the left and username and login link are on the right. Below that is a bar that is our top navigation bar, with some options that apply to the article the user is currently viewing. Along the left side is another navigation bar, this one giving the user access to some specific articles or views. To the right of that is the area where the current article is shown, and below all that is a bottom footer section that has a link to the DWR site, or a countdown timer showing the remaining time on an article lock when editing an article.

With that picture in mind, we can go forward and see how it's all put together. We'll begin with the style sheet the application's UI look and feel is built on.

styles.css

The `styles.css` file is the style sheet for the DWiki application, and it's a perfectly mundane style sheet. I've chosen not to list it out here in order to save some space, but please do peruse it before continuing, and try not to fall asleep as you do!

This style sheet begins by using the `* { }` trick to define a global style for all our text. Beyond that, you have styles for the top section of the page above the top navigation bar, styles for the top navigation bar itself, styles for the left-side navigation bar, styles for where the articles themselves appear, styles for the footer at the bottom, styles for links, and one style for the links on the top navigation and left-side navigation bars when they are clickable and being hovered over. Truly, if you've seen a single style sheet before in your life (which you have if you read the previous chapter), there are no surprises to be found here.

index.jsp

The `index.jsp` page is the core of DWiki on the client in terms of the user interface. Surprisingly, there's really very little to it other than the basic structure of the page, and you can see that for yourself in Listing 5-7.

Listing 5-7. *The Main Markup for DWiki,* `index.jsp`

```
<html>

  <head>

    <title>DWIKI - A DWR-Based Wiki</title>

    <link rel="stylesheet" href="css/styles.css" type="text/css">

    <script src="dwr/engine.js"></script>
    <script src="dwr/util.js"></script>
    <script src="dwr/interface/ArticleDAO.js"></script>

    <script src="js/RolloversClass.js"></script>
    <script src="js/DWikiClass.js"></script>

  </head>

  <body onLoad="DWiki.init();">

    <table width="90%" border="0" align="center" cellpadding="0"
      cellspacing="0">

      <tr>
        <td class="cssTop">
          <img src="img/logo.gif">
        </td>
        <td align="right" class="cssTop" valign="top">
```

```
    <span id="userInfo">
      <%
        if (request.getUserPrincipal() != null) {
      %>
        <%=request.getUserPrincipal().getName()%>
      <%
        } else {
      %>
        Anonymous User
      <%
        }
      %>
    </span>
    |
    <a href="loginOk.jsp" target="_new">Login</a>
  </td>
</tr>

<tr>
  <td colspan="2" class="cssTopNav" align="center">
    <img src="img/article0.gif" id="article" class="cssClickable"
      onClick="DWiki.articleClicked();"
      onMouseOver="Rollovers.imgOver(this);"
      onMouseOut="Rollovers.imgOut(this);">
    <img src="img/edit0.gif" id="edit" class="cssClickable"
      onClick="DWiki.editClicked();"
      onMouseOver="Rollovers.imgOver(this);"
      onMouseOut="Rollovers.imgOut(this);">
    <img src="img/history0.gif" id="history" class="cssClickable"
      onClick="DWiki.historyClicked();"
      onMouseOver="Rollovers.imgOver(this);"
      onMouseOut="Rollovers.imgOut(this);">
    <img src="img/comments0.gif" id="comments" class="cssClickable"
      onClick="DWiki.commentsClicked();"
      onMouseOver="Rollovers.imgOver(this);"
      onMouseOut="Rollovers.imgOut(this);">
  </td>
</tr>

<tr>
  <td colspan="2">

    <table border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td valign="top" class="cssLeftNav">
          <img src="img/frontPage0.gif" id="frontPage"
            class="cssClickable" onClick="DWiki.getArticle('FrontPage');"
```

```
        onMouseOver="Rollovers.imgOver(this);"
        onMouseOut="Rollovers.imgOut(this);"><br>
      <img src="img/allArticlesIndex0.gif" id="allArticlesIndex"
        class="cssClickable"
        onClick="DWiki.getArticle('AllArticlesIndex');"
        onMouseOver="Rollovers.imgOver(this);"
        onMouseOut="Rollovers.imgOut(this);"><br>
      <img src="img/search0.gif" id="search"
        class="cssClickable"
        onClick="DWiki.getArticle('Search');"
        onMouseOver="Rollovers.imgOver(this);"
        onMouseOut="Rollovers.imgOut(this);"><br>
      <img src="img/help0.gif" id="help"
        class="cssClickable" onClick="DWiki.getArticle('DWikiHelp');"
        onMouseOver="Rollovers.imgOver(this);"
        onMouseOut="Rollovers.imgOut(this);"><br>
    </td>

    <td width="100%" class="cssArticle" valign="top">

      <span id="loadingMessage">Loading...</span>

      <span id="articleContents" style="display:none;"></span>

      <span id="articleEditing" style="display:none;">
        You are now editing the article
        entitled: <b><span id="articleEditing_title"> </span></b>
        <br><br>
        <textarea id="articleEditing_text"
          cols="80" rows="10"></textarea>
        <br><br>
        <input type="button" value="Click To Save Changes"
          onClick="DWiki.updateArticle();">
      </span>

      <span id="articleHistory" style="display:none;"></span>

      <span id="articleComments" style="display:none;"></span>

    </td>
  </tr>

  <tr valign="middle">
    <td height="22" colspan="2" class="cssFooter" align="center"
      id="statusBar">
      Powered By <a href="http://getahead.org/dwr">DWR</a>
```

```
            </td>
          </tr>

        </table>
      </td>
    </tr>

  </table>

</body>

</html>
```

After the import of the style sheet, we find imports for DWR. Aside from the standard `engine.js` import, we also see that we're pulling in `util.js`, which we'll need for a couple of utility functions. After that is the import of the client-side proxy code for the `ArticleDAO` class. Following that are the two imports for the `DWikiClass` and `RolloversClass` JavaScript that makes up DWiki. And with that, the `<head>` section of our document is complete.

Moving on, we come to the `<body>`, and the first thing you'll notice is that `onLoad` there is a call to the `init()` method of the `DWikiClass` instance (pointed to by the variable `DWiki`, which is created in the `DWikiClass.js` file). We'll see what that method does shortly, but for now suffice it to say that it sets everything up for the application to be usable at startup.

The rest of the body of the document is pretty much straightforward markup building the screen as it appears. A small bit of Java scriplet code is present as well, and its job is to render the name of the user. The first time the user accesses the page, "Anonymous User" will be displayed in the upper-right corner. As you can see, this is accomplished by calling `request.getUserPrincipal()`, and if it's `null`, meaning the user hasn't yet logged in, he or she is anonymous. If it's not null, a call to `request.getUserPrincipal().getName()` is made, which returns the name of the user logged in, and we display that. A tiny bit of personalization can go a long way to making an application seem friendlier to the user, which makes the user like using your application a little more.

Next to the username you'll notice the link for logging in, and you'll also notice that it targets a new window. Let's discuss the login process now.

To log in, as was previously discussed, we need to request the protected resource `loginOk.jsp`, which results in `login.jsp` being served. Being an Ajax-based Web 2.0 application, we wouldn't feel right about reloading the page now would we? So, instead, we open the login "flow," so to speak, in a new window. When the login is successful, we'll simply need to update the name displayed in the upper-right corner, and that's it. All subsequent requests will effectively be logged-in requests.

Next, we find the markup for the top navigation menu. You'll notice these are all simple `<img>` tags. Each one has an `onClick` event handler attached, and all of them call methods in the `DWikiClass` instance. The methods are simply the name of the button, plus the word "Clicked": `articleClicked()`, `editClicked()`, `historyClicked()`, and `commentsClicked()`. You'll also notice that each image has `onMouseOver` and `onMouseOut` event handlers attached, and they make use of the `imgOver()` and `imgOut()` methods in the `RolloversClass` instance (pointed to by the `Rollovers` variable). These methods take as arguments a reference to a particular image. The method uses the ID of the `<img>` tag to do a typical JavaScript rollover effect. There's nothing too out of the ordinary here.

After that is the markup for the left navigation menu, and you'll see that each item is essentially just like those in the top menu bar. The methods called `onClick` for each are of course different, but you'll notice they're all the same: `getArticle()`. You see, each of those items, even though they are in some cases dynamically generated (like the all articles index, for example), are treated just like any other article is. This was a design decision that greatly simplified making these things work. As far as the UI is concerned, it's simply displaying an article; it has no idea that it may be something rather different than an article a user might create (such as the search form, which obviously isn't a typical article).

The main article content area follows, and here you'll notice there are actually a few different `<span>`s in play. The first is the loading message you see at various times when the server is being called upon. The second is where an article is actually inserted. The third is seen when you are editing an article. The fourth is where history is displayed, and the fifth is where comments are displayed. I originally had intended to have only a single `<span>`, and all of this content would be overwritten whenever appropriate; but as it turns out, that was more difficult to implement and would require more hits on the server, so this design made more sense in terms of simplicity and performance, even though that seems, to me anyway, like the opposite of what should be the case at first.

Things close out here with the footer section, and that's nothing but a link to the DWR web site (credit where credit's due, right?). This content is overwritten when an article is locked for editing so the time countdown can be shown, but it is immediately returned to what you see here when the lock expires.

That covers the main markup of DWiki. The form and structure of the page is, I think, pretty obvious, and there's nothing here that should be too surprising to you. With that out of the way, let's take a look at the two JSPs that deal with the login "flow," such as it is!

login.jsp

The login page is where we hook into the container-managed security in a sense. Recall that we specified the `FORM` auth method in `web.xml`, so when a constrained resource is requested, the user will be redirected to the specified login page; the code for this can be seen in Listing 5-8.

Listing 5-8. *The Login Page Markup, As Found in the File* `login.jsp`

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <img src="img/logo.gif" border="0">
    <br><br>
    <%
      if (request.getParameter("login_bad") != null) {
    %>
        <font color="#ff0000">Login invalid.  Please try again.</font>
```

```
    <%
      } else {
    %>
        Please log in to DWiki:
    <%
      }
    %>
    <br><br>
    <form name="j_security_check" method="post" action="j_security_check">
      <table border="0" cellpadding="2" cellspacing="2">
        <tr>
          <td width="1">Username: </td>
          <td><input type="text" name="j_username" value="" size="11"
            maxlength="10"></td>
        </tr>
        <tr>
          <td>Password: </td>
          <td><input type="password" name="j_password" value="" size="11"
            maxlength="10"><br>
        </tr>
        <tr>
          <td colspan="2" align="right">
            <input type="submit" value="Login">
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

This is a perfectly typical login page, which you can see in Figure 5-4. However, you'll notice three things about it that will be new if you have never dealt with J2EE security before. First, note the action of the form, `j_security_check`. This is a servlet that the container provides, and its job is to validate the credentials entered by the user. Those credentials must be submitted via parameters with the names `j_username` and `j_password`. When the servlet is invoked by the form submission, if the credentials are valid, the servlet redirects to the originally requested resource, which in our case is `loginOk.jsp`.

You'll also notice the JSP section shortly after the opening `<body>` tag. Recall in `web.xml` that the specified login error page was `login.jsp`, and the URL had a query string with a single parameter, `login_bad`. When that parameter is present when `login.jsp` is rendered, a string saying the entered information was invalid is shown. When it's not present, it means this is an attempted login rather than a failed attempt, and the polite prompt to log in is instead rendered. All of this is accomplished in this code block.

Figure 5-4. *The DWiki login page*

loginOk.jsp

When the user submits valid credentials from `login.jsp`, the page `loginOk.jsp` is returned. It isn't much to look at, I admit, so I've saved some space by not showing it here, and its code isn't all that much more to look at really, as shown in Listing 5-9; it's only marginally more interesting than seeing it onscreen.

Listing 5-9. *The Response to a Login, Rendered by* `loginOK.jsp`

```
<html>
<head>
  <title>DWiki Login Successful</title>
  <script>
    function processLogin() {
      opener.document.getElementById("userInfo").innerHTML =
        "<%=request.getUserPrincipal().getName()%>";
    }
  </script>
</head>
<body onLoad="processLogin();">
  <img src="img/logo.gif" border="0">
  <br><br>
  You have been successfully logged in to DWiki.  You may now create and edit
  articles.  Please click
  <a href="javascript:void(0);" onClick="window.close();">HERE</a>
  to close this window and return to DWiki.
</body>
</html>
```

On load, the `processLogin()` function is called. This function takes the username that the user logged in under, as returned by `request.getUserPrincipal.getName()`, and inserts it into the `<div>` on the parent page. This is in the upper-right corner, where "Anonymous User" would previously have been. Again, just a little personal touch that vain humans tend to like.

RolloversClass.js

The next source file to explore is `RolloversClass.js`, which defines the JavaScript `RolloversClass` class. Listing 5-10 shows this source code.

Listing 5-10. *The* `RolloversClass` *Source File*

```
function RolloversClass() {

  var frontPage0 = new Image(113, 22);
  frontPage0.src = "img/frontPage0.gif";
  var frontPage1 = new Image(113, 22);
  frontPage1.src = "img/frontPage1.gif";
  var allArticlesIndex0 = new Image(113, 22);
  allArticlesIndex0.src = "img/allArticlesIndex0.gif";
  var allArticlesIndex1 = new Image(113, 22);
  allArticlesIndex1.src = "img/allArticlesIndex1.gif";
  var search0 = new Image(113, 22);
  search0.src = "img/search0.gif";
  var search1 = new Image(113, 22);
  search1.src = "img/search1.gif";
  var help0 = new Image(113, 22);
  help0.src = "img/help0.gif";
  var help1 = new Image(113, 22);
  help1.src = "img/help1.gif";
  var article0 = new Image(113, 22);
  article0.src = "img/article0.gif";
  var article1 = new Image(113, 22);
  article1.src = "img/article1.gif";
  var edit0 = new Image(113, 22);
  edit0.src = "img/edit0.gif";
  var edit1 = new Image(113, 22);
  edit1.src = "img/edit1.gif";
  var history0 = new Image(113, 22);
  history0.src = "img/history0.gif";
  var history1 = new Image(113, 22);
  history1.src = "img/history1.gif";
  var comments0 = new Image(113, 22);
  comments0.src = "img/comments0.gif";
  var comments1 = new Image(113, 22);
  comments1.src = "img/comments1.gif";
```

```
  this.imgOver = function(inImg) {

    var imgName = inImg.id;
    var img1 = eval(imgName + "1");
    inImg.src = img1.src;

  } // End imgOver().

  this.imgOut = function(inImg) {

    if (inImg.id != DWiki.currentMode) {
      var imgName = inImg.id;
      var img0 = eval(imgName + "0");
      inImg.src = img0.src;
    }

  } // End imgOut().

} // End class.


// The one and only instance of RolloversClass.
var Rollovers = new RolloversClass();
```

It begins unassumingly enough by preloading a batch of images. These are the images for our top and left navigation bars. This gives us nice, smooth rollover effects when the user mouses over the items (otherwise, the images would be loaded upon that event firing, which would cause a noticeable delay, at least the first time, while the alternate hover images were loaded).

Following the preloads are two simple functions, `imgOver()` and `imgOut()`, fired correspondingly in response to the `onMouseOver` and `onMouseOut` events hooked to the navigation items. Each one works essentially the same. First, a simple check to make sure the item being hovered over isn't for the current mode (the possible modes, which we'll see later, are viewing an article, editing an article, viewing an article's history, and viewing an article's comments). Assuming it's not the current mode (which means it's already highlighted and there's nothing to do right now), we get the ID of the object passed in (which is the image itself). We then use the `eval()` function to get a reference to the appropriate preloaded image, either for the hover image in the case of `imgOver()` or the normal image for `imgOut()`. We then simply change the `src` attribute of the image passed in to that of the preloaded image, and we have ourselves a rollover (although not the more, err, exciting, type of rollover as seen frequently in the better NASCAR races).

The final line of the file instantiates the class and stores the reference in the `Rollovers` variable, and that's a wrap on rollovers.

DWikiClass.js

Now we come to the real heart of DWiki on the client, namely the `DWikiClass.js` file, which contains the `DWikiClass` class. Due to the relatively large size of this code, it isn't listed in its entirety here. Instead, I'll call out portions of it as we go through it when necessary.

As you start looking through this code (which I hope you're doing as you read this), you'll encounter the `DWikiClass` class, which, being JavaScript, is actually just a function. Within it you'll first find that there are a couple of public members (remember that in JavaScript, a variable is defined within a function using the keyword `this` before it is considered public, whereas one defined using `var` is essentially private). The first of these is `currentArticle`, and this stores a reference to an object that is basically the client-side version of the server-side `Article` class. When we retrieve an article, an `Article` object is created on the server and returned by DWR, which as we know will result in a JavaScript object representing that `Article` object. So, `currentArticle` stores the reference to that object.

Next is `currentMode`. There are four modes DWiki can be in. The first is when viewing an article (this mode is called "article"). The second is editing an article ("edit"), the third is viewing article history ("history"), and the last is viewing article comments ("comments"). This variable tells us which mode we're currently in.

After that is `lockTimer`. This is a reference to a JavaScript timer, which is used to display the lock timeout in the footer. Going along with that is `lockCountdown`, which is the number of seconds remaining before the lock expires. Also kind of related to edit timeout is the `statusBarMarkup`, which is the markup that starts out in the footer. This is used to replace the countdown when the lock expires.

After that begins the methods of the class, the first being `init()`. This is called when the page loads, and its sole function is to load the front page, which is the initial article the user sees. This is accomplished with a call to `getArticle()`, which we'll go out of order a bit here and look at now:

```
this.getArticle = function(inArticleTitle, inExtraData1, inExtraData2) {

  this.showLoading();
  ArticleDAO.getArticle(inArticleTitle, inExtraData1, inExtraData2, true,
    {
      callback : function(inResp) {
      DWiki.currentArticle = inResp;
      DWiki.switchMode("article");
    }
    }
  );

} // End getArticle().
```

As you'll recall from our look at `index.jsp`, let's say we click the Help link as an example. That calls `getArticle()`. The first argument is the title of the article, which is how articles are looked up. The other two parameters, `inExtraData1` and `inExtraData2`, are used when performing searches only. The first is the text to search for, and the second is what to search, be it the title of articles, the text of articles, or both.

This is actually a very simple function. First, we see a call to `showLoading()`, which is a simple function that hides the `<div>`s corresponding to the four modes (remember from `index.jsp` that each mode basically is housed in its own `<div>`), and then the "please wait" message is shown. Next is the first DWR call we've seen in this application. The `ArticleDAO` class, which we'll look at later, is the main workhorse on the server side, and in this case we're calling its `getArticle()` method, as you'd expect. Note the Boolean parameter, which here is set to `true`. This specifies whether article links should be expanded into clickable links. There are some cases where the `ArticleDAO.getArticle()` method is called where we don't want those links expanded (creating history records and locking the record for editing), so we need the ability to turn that function on and off, and that parameter gives us that capability.

The DWR callback has a simple job to do: set the `currentArticle` field to what was returned by the method, which is an `Article` object marshaled into a corresponding JavaScript object, and then a call to the `switchMode()` method to put the app into the appropriate mode for viewing the article retrieved.

The `switchMode()` method is actually a very important one, so let's have a look at it now:

```
this.switchMode = function(inMode) {

  this.currentMode = "";
  Rollovers.imgOut(dwr.util.byId("article"));
  Rollovers.imgOut(dwr.util.byId("edit"));
  Rollovers.imgOut(dwr.util.byId("history"));
  Rollovers.imgOut(dwr.util.byId("comments"));
  dwr.util.byId("articleContents").style.display = "none";
  dwr.util.byId("articleEditing").style.display = "none";
  dwr.util.byId("articleHistory").style.display = "none";
  dwr.util.byId("articleComments").style.display = "none";
  dwr.util.byId("loadingMessage").style.display = "none";
  this.currentMode = inMode;

  if (inMode == "article") {
    dwr.util.byId("articleContents").style.display = "";
    var lastEdited = "";
    // Insert last updated line, if present (for a new article, it wouldn't).
    if (this.currentArticle.lastEdited != null) {
      lastEdited = "<br><hr><br>Last edited " +
        this.currentArticle.lastEdited +
        " by " + this.currentArticle.lastEditedBy;
    }
    dwr.util.byId("articleContents").innerHTML = "<pre>" +
      this.currentArticle.text + "</pre>" + lastEdited;
  } else if (inMode == "edit") {
    dwr.util.byId("articleEditing").style.display = "";
  } else if (inMode == "history") {
    dwr.util.byId("articleHistory").style.display = "";
  } else if (inMode == "comments") {
    dwr.util.byId("articleComments").style.display = "";
```

```
    }
    Rollovers.imgOver(dwr.util.byId(this.currentMode));

  } // End switchMode().
```

It begins by resetting the rollover states of all the mode buttons on the top navigation bar. It also hides the s for all the modes. Note that even before that, it sets currentMode to a blank string. This is necessary because the rollover functions will use that value to highlight the link for the current mode, and we don't want that in this case. So, currentMode is set to a blank string before, and immediately after it is set to the mode specified to switch to by the parameter inMode. Then, some branching is done based on that specified mode. For the article viewing mode ("article"), some extra work is needed. First, the appropriate is shown for the mode. Next, the information line at the bottom that states when the article was last edited and by whom it is constructed. Finally, the article itself, in addition to the constructed information line string, is inserted into the . Note that the article contents are also wrapped in <pre> tags so that line breaks work without the having to explicitly put them in with markup, or us having to have written code to put them in somehow!

For the other modes, all that's required is to show the appropriate mode , nothing more. And finally, we use the RolloversClass instance to highlight the appropriate mode navigation line up top.

We skipped the addArticle() method previously, so let's go back to that now:

```
  this.addArticle = function(inArticleTitle) {

    if (dwr.util.getValue("userInfo").replace(/^\s*(.*\S|.*)\s*$/, '$1') ==
      "Anonymous User") {
      alert("You must be a registered user to add this article");
    } else {
      ArticleDAO.addArticle(inArticleTitle,
        {
          callback : function(inResp) {
            DWiki.getArticle(inArticleTitle);
          }
        }
      );
    }

  } // End addArticle().
```

Once more, we see DWR's uncanny ability to take something that might otherwise be a bit hairy and make it ridiculously simple! The first thing done is a quick check: since only registered users are allowed to create articles, the text currently in the upper-right corner, the username, is interrogated. If the value is "Anonymous User," the user cannot continue and is told so. Note the regular expression used to trim whitespace from both ends of the string. In some browsers, the call to dwr.util.getValue() will result in whitespace, which would cause a false mismatch in some cases, so this deals with that possibility (recall that the dwr.util.getValue() function gets the value of the specified element, regardless of what type it is. In this case, it's a 's innerHTML property).

Assuming the user can create the article, we have another simple call to the `ArticleDAO`'s `addArticle()` method. This time, all that's required is the title of the article being created, which is passed in, and which is the text of the link that was clicked to get here, since that's the only way an article can be created. The callback simply makes a call to `getArticle()`, so that the newly created article will be shown immediately. Sure, the article itself could have been returned by the call to `addArticle()`, which would have been slightly more efficient since it's one less call, but this serves to demonstrate DWR a little better, and also keeps the code on the server side a little less verbose.

The next batch of methods are all related to the top navigation buttons, beginning with `articleClicked()`. Obviously, this is called when the Article link is clicked, meaning the user wants to view the article. It's exceedingly simple:

```
this.articleClicked = function() {

  if (this.currentMode != "article") {
    this.switchMode("article");
  }

} // End articleClicked().
```

So long as the current mode is not already "article", we call the `switchMode()` method and switch to "article" mode. It's as simple as that!

Next up is `editClicked()`, which is a bit more involved:

```
this.editClicked = function() {

  if (this.currentArticle.title != "DWikiHelp" &&
    this.currentArticle.title != "AllArticlesIndex" &&
    this.currentArticle.title != "Search" &&
    this.currentArticle.title != "SearchResults") {
    if (dwr.util.getValue("userInfo").replace(/^\s*(.*\S|.*)\s*$/, '$1') ==
      "Anonymous User") {
      alert("You must be a registered user to edit this article");
    } else {
      if (this.currentArticle.articleExists == false) {
        alert("This article must be created before it can be edited.  " +
          "Click the link below to create it.");
        return;
      }
      if (this.currentMode != "edit") {
        this.showLoading();
        ArticleDAO.lockArticleForEditing(this.currentArticle.title,
          {
            callback : function(inResp) {
              if (inResp.indexOf("ok") != -1) {
                DWiki.lockCountdown = inResp.split("=")[1];
                DWiki.switchMode("edit");
                dwr.util.byId("articleEditing_title").innerHTML =
                  DWiki.currentArticle.title;
```

```
                document.getElementById("articleEditing_text").value =
                  inResp.split("=")[2];
                DWiki.lockTimer = setTimeout("DWiki.updateLockTimer()", 0);
              } else if (inResp.indexOf("lockedBy" != -1)) {
                var respArray = inResp.split("=");
                DWiki.switchMode(DWiki.currentMode);
                alert("Article is already locked by " + respArray[1] +
                  " for " + respArray[2] + " more seconds");
              } else {
                DWiki.switchMode(DWiki.currentMode);
                alert("Article could not be locked for editing, " +
                  "reason unknown");
              }
            }
          }
        );
      }
    }
  } else {
    alert("This function is not available for this article");
  }

} // End editClicked().
```

To begin, this function does a check that you'll see a few other times. Recall that the help view, the list of all articles on the wiki, the search view, and, of course, search results are all basically just articles to DWiki. However, these articles are a bit different in that they aren't editable by users. Therefore, before we allow the Edit link to do its thing, we need to make sure it's not one of those articles currently being viewed. Assuming it's not, we then check to be sure the user is a registered user in the same way we saw previously. Assuming he or she is, we then check to see whether the article already exists or not by checking the `articleExists` property of the `Article` object being pointed to by the `currentArticle` variable (remember that DWR returns a JavaScript representation of an `Article` object, and we store the reference to it in `currentArticle`). Assuming the article already exists, we then check to be sure we're not currently in "edit" mode already. Assuming we aren't, we realize we've made a boatload of assumptions! But, in the end, they work out, and we can begin editing the article.

To do so, we first need to lock the article, or more precisely, see whether it's already being edited. To do that, we call the `lockArticleForEditing()` method of the `ArticleDAO` object. The return value from this method is a string, and it can be one of two things. If the word "lockedBy" is in the string, the string is in the form `lockedBy=xxx`, where `xxx` is the name of the user who has the article locked. We use this to pop up a dialog box informing the user that he or she can't edit the article and telling that user who has it locked. If the article isn't locked, the string is in the form `ok=xxx=yyy`, where `xxx` is the number of seconds the edit lock will last, and `yyy` is the text of the article with no link expansion done. When we get the "ok" string, it's split into its component parts, and we switch the mode to "edit" via a call to `switchMode()`. Next, we set the `innerHTML` of the `<div>` where the title of the article is displayed on the edit page to the title, which we get from the `currentArticle` reference. Next, the value of the `<textarea>` where

the user can edit the article is populated with the text returned by the `lockArticleForEditing()` method. Lastly, we begin the countdown timer to display how much of the edit lock remains.

To be safe, we also have a branch that covers any sort of unexpected response. It simply switches back to whatever the mode was before the edit link was clicked.

If you have been thinking along about how you might have implemented things, you may now be questioning why I didn't use the `dwr.util.setValue()` function to set the value of the `<textarea>`. The answer is that that function seems to expand any characters it recognizes as having HTML entities into those entities. So, a `<` becomes `<` in the `<textarea>`. Unfortunately, since we allow HTML to be present in articles, this is essentially a bug as far as DWiki is concerned, at least, that's how a user would view it. Therefore, I set the value directly instead to avoid this problem.

Remember that timer we just set in `editClicked()`? Looking at the code, you'll realize that the function it executes is `updateLockTimer()`, which is what we're ready to look at now.

```
this.updateLockTimer = function() {

  if (DWiki.lockCountdown == 0) {
    dwr.util.byId("statusBar").innerHTML = DWiki.statusBarMarkup;
    alert("Your edit lock on this article has expired.  Your changes will " +
      "be lost.");
    DWiki.lockTimer = null;
    return;
  }
  dwr.util.byId("statusBar").innerHTML = "Edit lock will expire in " +
    DWiki.lockCountdown + " seconds";
  DWiki.lockCountdown = DWiki.lockCountdown - 1;
  DWiki.lockTimer = setTimeout("DWiki.updateLockTimer()", 1000);

} // End updateLockTimer().
```

So, it's pretty simple. First, we check to see whether the edit lock has expired, which we do by seeing whether the `lockCountdown` field has hit zero. If so, we redisplay the DWR home page link in the footer and inform the user that his or her lock has expired. If the lock hasn't yet expired, we update the display in the footer and reduce `lockCountdown` by 1. Finally, we fire the timeout again, one second later, to continue counting down.

The `historyClicked()` method is the next method you'll find as you walk through the `DWikiClass.js` file, and it's of course called when the history link is clicked.

```
this.historyClicked = function() {

  if (this.currentArticle.title != "DWikiHelp" &&
    this.currentArticle.title != "AllArticlesIndex" &&
    this.currentArticle.title != "Search" &&
    this.currentArticle.title != "SearchResults") {
```

```
    if (this.currentMode == "history") {
      return;
    }
    this.showLoading();
    ArticleDAO.getArticleHistory(this.currentArticle.title,
      {
        callback : function(inResp) {
          DWiki.switchMode("history");
          dwr.util.byId("articleHistory").innerHTML = inResp;
        }
      }
    );
  } else {
    alert("This function is not available for this article");
  }

} // End historyClicked().
```

There's that check once again to make sure the user isn't trying to edit an article that isn't editable, such as the faux articles generated by the application for article history, for example. Also again is the check to be sure we aren't already in this mode. If both those checks are passed, we call the `getArticleHistory()` method of `ArticleDAO`. This method returns the markup that is to be displayed showing the history of the article. We switch to the "history" mode, and we're good to go.

Whoa, not so fast! One of the things you can do in the history display is to click a history item to see the text of the article before and after the edit. When you click one of the history items, you wind up calling the `toggleHistory()` method of `DWikiClass`, which is this code:

```
this.toggleHistory = function(inID) {

  var historyDiv = dwr.util.byId(inID);
  if (historyDiv.style.display == "none") {
    historyDiv.style.display = "";
  } else {
    historyDiv.style.display = "none";
  }

} // End toggleHistory().
```

It's just a simple matter of getting a reference to the `<div>` corresponding to the history item clicked, and either showing or hiding it, whatever the opposite of its current state is.

Next up is the `commentsClicked()` method, which I'm going to skip showing here because it is almost identical to the `historyClicked()` method. One thing different though is that when we add a comment, we're obviously in "comments" mode, so we need to refresh it to see the newly added comment. To accomplish this, there is an `inForceUpdate` parameter to this method. When true, it will cause the mode switch to occur, even if we're already in "comments" mode, which has the effect of refreshing the display.

Speaking of adding comments, the `addComment()` method actually follows this one, and it's very simple:

```
this.addComment = function() {

  ArticleDAO.addComment(
    {
      articleTitle : this.currentArticle.title,
      text : dwr.util.getValue("commentText")
    },
    {
      callback : function(inResp) {
        DWiki.commentsClicked(true);
      }
    }
  );

} // End addComment().
```

We've already seen here how DWR marshals Java objects to JavaScript objects (the `Article` objects being returned), but do you remember how to send an object to Java **from** JavaScript? When we look at it later, you'll see that the `addComment()` method of the `ArticleDAO` object takes as an argument an `ArticleComment` object. This is a simple javabean class with four fields: the article title, the text of the comment, the date/time the comment was posted, and the name of the user who posted it. Now, the last two will be filled out automatically during the comment save process, but the first two must be sent in, so we basically need to construct an equivalent JavaScript object to the Java version of `ArticleComment`. That's a simple matter of using some object notation as the argument:

```
{
  articleTitle : this.currentArticle.title,
  text : dwr.util.getValue("commentText")
}
```

That's all it takes! After the comment is saved, the callback function simply does the equivalent of the user clicking the comments link, passing in that parameter to force the refresh, and voilá, the new comment is shown.

We're now down to the last method in the `DWikiClass` class, and that's `updateArticle()`.

```
this.updateArticle = function() {

  if (this.lockTimer == null) {
    alert("Your lock on this article has expired.  You will need to " +
      "re-acquire it by clicking the Article link above, then clicking the " +
      "Edit link again");
    return;
  }
  this.currentArticle.text = dwr.util.getValue("articleEditing_text");
  ArticleDAO.updateArticle(this.currentArticle, true,
```

```
      {
        callback : function(inResp) {
          clearTimeout(DWiki.lockTimer);
          DWiki.lockTimer = null;
          dwr.util.byId("statusBar").innerHTML = DWiki.statusBarMarkup;
          DWiki.getArticle(DWiki.currentArticle.title);
        }
      }
    );

  } // End updateArticle().
```

First, we need to ensure that the edit lock the user holds on the current article hasn't yet expired, so as long as there is still a value in the `lockTimer` field, we're good to go. After that, we grab the text that the user has entered and update the `currentArticle` instance with it, and we call the `updateArticle()` method of the `ArticleDAO` class, passing the `currentArticle` instance to it, along with the second parameter set to `true` (which, as you'll see later, causes a history record to be written out for the article). The callback is just some simple cleanup of the edit lock timer stuff, plus retrieving the article, which switches to "article" mode. All in all, pretty easy, right?

FreeMarker Templates

The next piece of the puzzle is to look at the FreeMarker templates that generate much of the output generated by the server-side code. We'll get into the details of how these templates are executed later; for now we're just interested in the templates themselves.

AllArticlesIndex.ftl

The `AllArticlesIndex.ftl` template, as shown in Listing 5-11, generates the list of all articles on the wiki.

Listing 5-11. *The* `AllArticlesIndex.ftl` *FreeMarker Template*

```
<b>All articles on DWiki (if any):</b><ul><#list articles as article><li>
<a href="javascript:void(0);"
  onClick="DWiki.getArticle('${article.TITLE}');">${article.TITLE}</a>
  Created by ${article.CREATOR} on ${article.CREATED}</li>
</#list></ul>
```

You can probably understand this just fine based on our earlier discussion of FreeMarker. It's simply an iteration over the collection of `Article` objects that is supplied when the template is executed. For each, we generate a link that calls the `DWiki.getArticle()` method, passing in the title of the article. The title is the text displayed in the list, and we also show the article's creator and when it was created.

In Figure 5-5, you can see the output of this template for yourself.

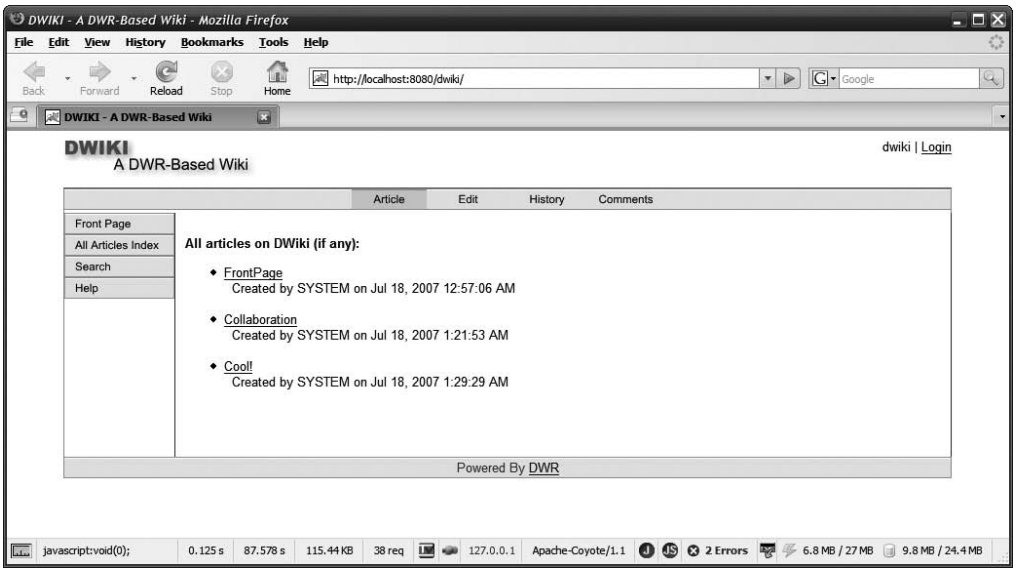


Figure 5-5. *The list of all articles on the wiki*

ArticleComments.ftl

The `ArticleComments.ftl` template generates the display to show comments for an article, as you can see in Figure 5-6.

The template itself is shown in Listing 5-12. This one is an iteration over the collection of `ArticleComment` items present in the data model handed to the template. For each, we're showing who posted it and when, and, of course, the text of the comment.

Listing 5-12. *The* `ArticleComments.ftl` *FreeMarker template*

```
Here are the comments for this article:<ul>
<#list commentItems as commentItem><li>
    Posted by ${commentItem.POSTER} on ${commentItem.POSTED}
  <br><br>
   ${commentItem.TEXT}
  </li><br>
</#list></ul>
<#if dwikiUser = true>
Add a comment:
<br>
<textarea id="commentText" cols="80" rows="10"></textarea>
<br>
<input type="button" value="Click To Add Comment"
  onClick="DWiki.addComment();">
</#if>
```

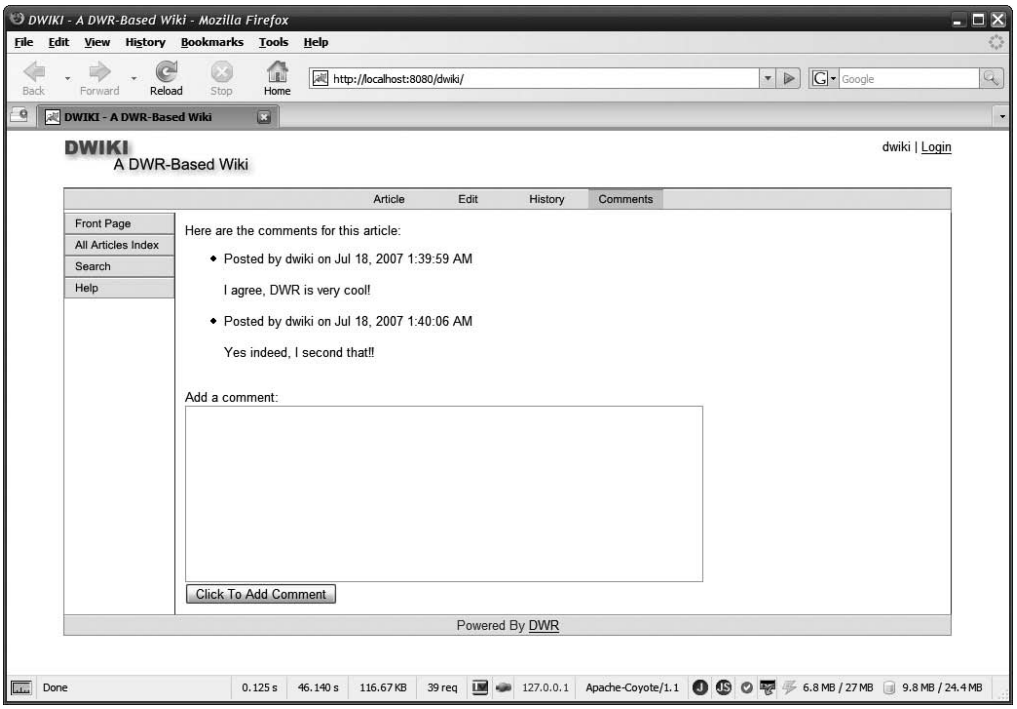


Figure 5-6. *The article comments view*

We're then using the branching logic capabilities of FreeMarker to only render the section where a new comment can be added if the user is a registered user, which is determined by the value of `dwikiUser` in the data model.

ArticleHistory.ftl

Next up is the template for showing the history of an article, as shown in Listing 5-13 and Figure 5-7.

Listing 5-13. *The* `ArticleHistory.ftl` *FreeMarker Template*

```
Here is the history for this article:<ul>
<#list historyItems as historyItem><li><a href="javascript:void(0);"
  onClick="DWiki.toggleHistory(
  '${historyItem.ARTICLETITLE}_${historyItem_index}');">
    Edited by ${historyItem.EDITEDBY} on ${historyItem.EDITED}
  </a><br><br>
  <div
    id="${historyItem.ARTICLETITLE}_${historyItem_index}" style="display:none";>
    <div style="background-color:#e0e0e0;">Previous Text:</div>
    ${historyItem.PREVIOUSTEXT}
    <br><br>
    <div style="background-color:#e0e0e0;">New Text:</div>
```

```
      ${historyItem.NEWTEXT}
    </div>
  </div>
  </li><br>
</#list>
```

This is yet another simple iteration, this time over a collection of `ArticleHistoryItem` objects. For each we're showing the date and time when the edit was made, and who made it. This is a clickable link, and below it are two `<div>`s, one showing the text of the article before the edit, the other showing the text after. When clicked, the `toggleHistory()` method that we looked at earlier is called, and these `<div>`s are shown and hidden in a toggling fashion (hence the creative method naming!).

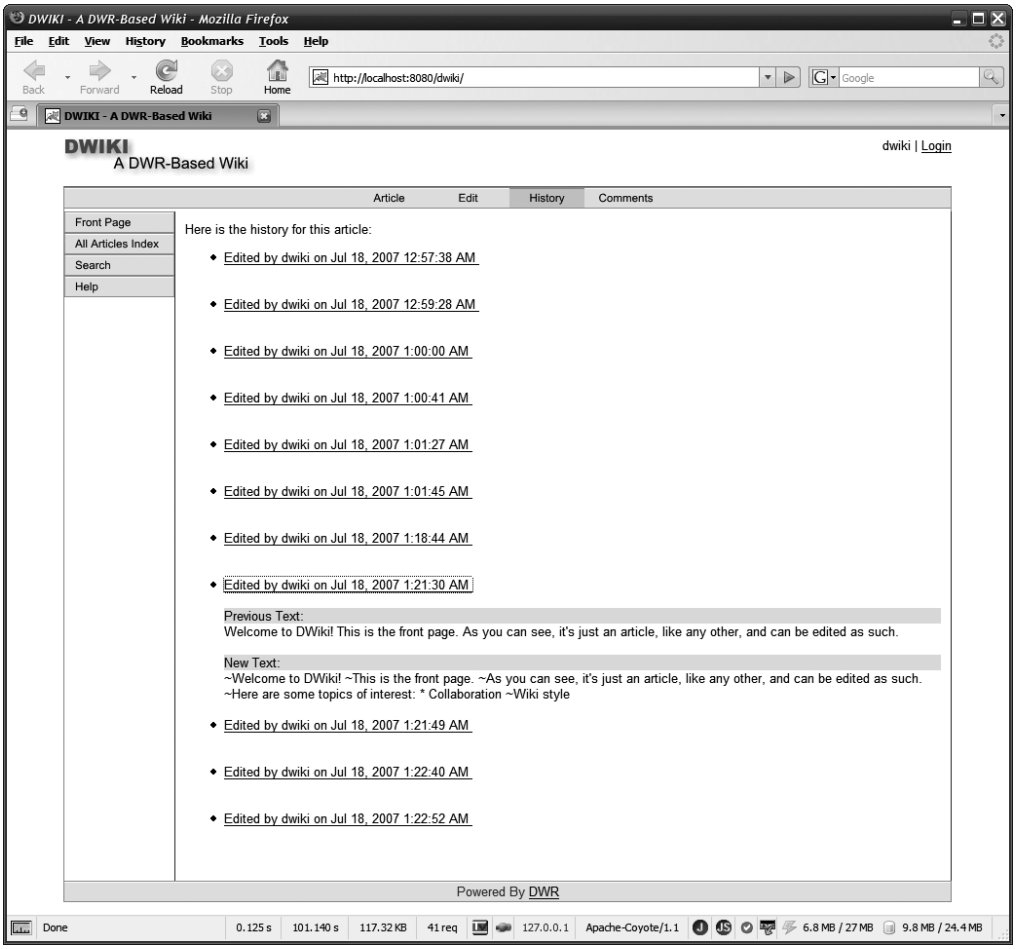

DWIKI - A DWR-Based Wiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/dwiki/
Google
DWIKI - A DWR-Based Wiki
DWIKI
A DWR-Based Wiki
dwiki | Login
Article Edit History Comments
Front Page
All Articles Index
Search
Help
Here is the history for this article:
Edited by dwiki on Jul 18, 2007 12:57:38 AM
Edited by dwiki on Jul 18, 2007 12:59:28 AM
Edited by dwiki on Jul 18, 2007 1:00:00 AM
Edited by dwiki on Jul 18, 2007 1:00:41 AM
Edited by dwiki on Jul 18, 2007 1:01:27 AM
Edited by dwiki on Jul 18, 2007 1:01:45 AM
Edited by dwiki on Jul 18, 2007 1:18:44 AM
Edited by dwiki on Jul 18, 2007 1:21:30 AM
Previous Text:
Welcome to DWiki! This is the front page. As you can see, it's just an article, like any other, and can be edited as such.
New Text:
~Welcome to DWiki! ~This is the front page. ~As you can see, it's just an article, like any other, and can be edited as such.
~Here are some topics of interest: * Collaboration ~Wiki style
Edited by dwiki on Jul 18, 2007 1:21:49 AM
Edited by dwiki on Jul 18, 2007 1:22:40 AM
Edited by dwiki on Jul 18, 2007 1:22:52 AM
Powered By DWR
Done
0.125 s
101.140 s
117.32 KB
41 req
127.0.0.1
Apache-Coyote/1.1
6.8 MB / 27 MB
9.8 MB / 24.4 MB

Figure 5-7. *The article history view*

DWikiHelp.ftl

The `DWikiHelp.ftl` template is the template that generates the help display. Because this one is a bit large, and really is nothing but plain text anyway, I've chosen not to show it here. Please have a look at it anyway before moving on.

newArticle.ftl

When an article has been created, but has not yet been edited, a simple display explaining this is shown; this display is generated by the `newArticle.ftl` template, as shown in Listing 5-14.

Listing 5-14. *The* `newArticle.ftl` *FreeMarker Template*

```
~This article has been created, but has not yet been edited.

~Click the ~EDIT link above to begin editing it.
```

When you actually click the Edit link, you are treated to the edit display shown in Figure 5-8, which we've actually already discussed.

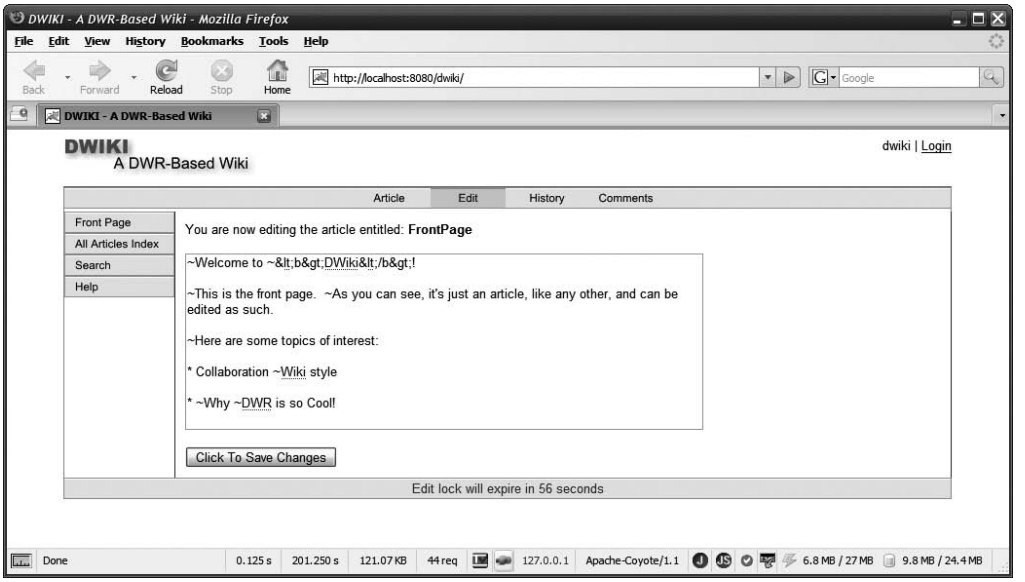

DWIKI - A DWR-Based Wiki - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/dwiki/
Google
DWIKI - A DWR-Based Wiki
DWIKI
A DWR-Based Wiki
dwiki | Login
Article Edit History Comments
Front Page
All Articles Index
Search
Help
You are now editing the article entitled: FrontPage
~Welcome to ~DWiki!
~This is the front page. ~As you can see, it's just an article, like any other, and can be edited as such.
~Here are some topics of interest:
* Collaboration ~Wiki style
* ~Why ~DWR is so Cool!
Click To Save Changes
Edit lock will expire in 56 seconds
Done
0.125 s
201.250 s
121.07 KB
44 req
127.0.0.1
Apache-Coyote/1.1
6.8 MB / 27 MB
9.8 MB / 24.4 MB

Figure 5-8. *Editing an article*

nonexistentArticle.ftl

When a link in an article is clicked that leads to an article that has not yet been created, the user is shown that, and given the opportunity to create the article.

In Listing 5-15, we can see the template, `nonexistentArticle.ftl`, which generates that display. The article title is inserted as the argument to the `addArticle()` method, called when the user clicks the HERE link, and that's how an article gets created on DWiki!

Listing 5-15. *The* `nonExistentArticle.ftl` *FreeMarker Template*

```
Page does not yet exist.  Click <a href="javascript:void(0);"
  onClick="DWiki.addArticle('${articleTitle}');">HERE</a> to create it.
```

Search.ftl

The user can search for articles on DWiki by clicking the Search navigation link on the left. This results in the display shown in Figure 5-9.

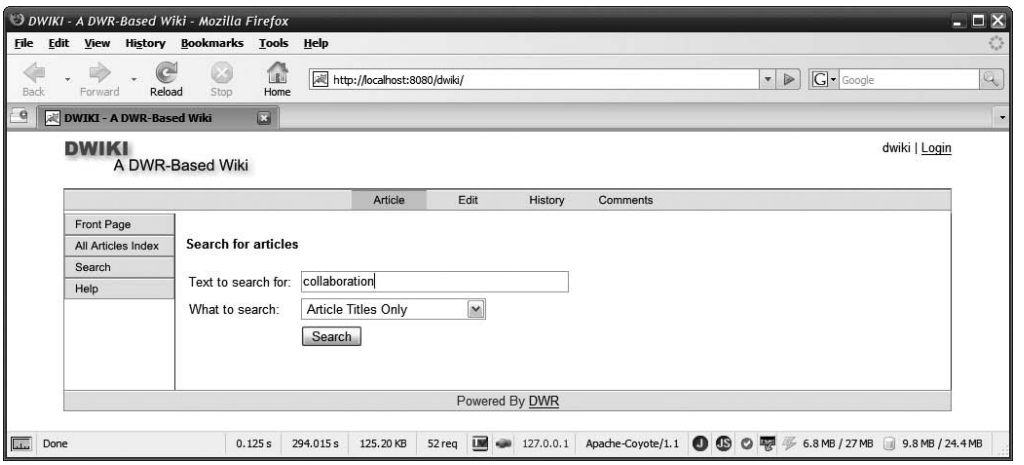


Figure 5-9. *The article search view*

The template that generates this display, which you'll note is actually nothing but straight markup, is that shown in Listing 5-16.

Listing 5-16. *The* `Search.ftl` *FreeMarker Template*

```
<b>Search for articles</b>

<table border="0" cellpadding="2" cellspacing="2">
  <tr>
    <td>Text to search for: </td>
    <td><input type="text" id="searchText" size="40"
      value=""></td>
  </tr>
  <tr>
    <td>What to search: </td>
    <td>
      <select id="searchWhat">
        <option value="title">Article Titles Only</option>
        <option value="text">Article Text Only</option>
        <option
          value="both">Both Article Titles And Text</option>
```

```
      </select>
    </td>
  </tr>
  <tr>
    <td> </td>
    <td><input type="button" value="Search"
      onClick="DWiki.getArticle('SearchResults',
      dwr.util.getValue('searchText'), dwr.util.getValue('searchWhat'));">
    </td>
  </tr>
</table>
```

Although it is indeed just simple HTML, making it a FreeMarker template keeps things consistent with the other noneditable views in DWiki and allows us to change it easily.

SearchResults.ftl

Finally, when a search is actually performed, the results of the search, as shown in Figure 5-10, are generated by the `SearchResults.ftl` template, the code of which is shown in Listing 5-17.

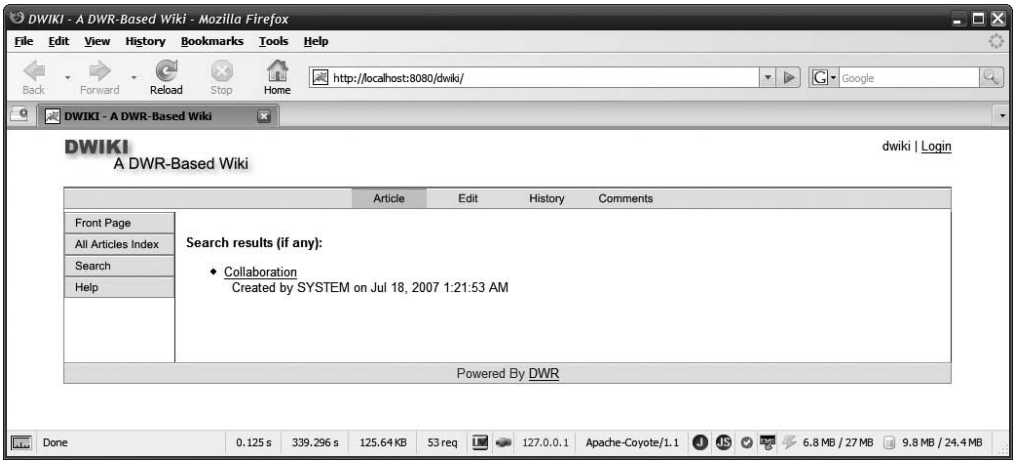


Figure 5-10. *The results of performing a search*

Listing 5-17. *The* `SearchResults.ftl` *FreeMarker Template*

```
<b>Search results (if any):</b><ul><#list articles as article><li>
<a href="javascript:void(0);"
  onClick="DWiki.getArticle('${article.TITLE}');">${article.TITLE}</a>
  Created by ${article.CREATOR} on ${article.CREATED}</li>
</#list></ul>
```

This template is just as simple as the rest. No logic even, just a simple iteration over a collection of `Article` objects. For each, we show the title, the creator, and the date created. The title is clickable, calling on the `getArticle()` method to show the article. Absolute piece of cake!

The Server-Side Code

We now come to our examination of the server-side code. There are some interesting things to see here, and it'll quickly become apparent how it all ties together, I think, the client-side code calling on this code, and so on. No point delaying, let's jump right in!

Config.java

The `Config` class stores the configuration information read in from `dwiki.properties`. It's a sublimely ordinary javabean, nothing too special. In fact, Figure 5-11, the UML diagram for this class, pretty well sums it up.

Figure 5-11. `Config` *class UML diagram*

The code itself, which I haven't listed here because it's rather lengthy, is nothing but the private members you see in the diagram, each corresponding to an item in the properties file, and their corresponding getters and setters. The setters are all static so that once the values are populated, other code can retrieve them quickly and easily. There is also an overloaded `toString()` method, my usual reflection-based version, so we can always display the contents of the object easily.

DWikiContextListener.java

When DWiki starts up, it has some initialization tasks to perform, namely loading the properties file and verifying that the database exists and is good to go. The `DWikiContextListener` class is what accomplishes these goals, and its UML diagram is shown in Figure 5-12.

Figure 5-12. `DWikiContextListener` *class UML diagram*

In Listing 5-18, you can see that this class is just a normal, everyday context listener, so it will be executed by the container when the context, that is, the application, starts up. Its work is done in the `contextInitialized()` method, beginning with the task of loading the properties file. Recall that the file is in the `WEB-INF/classes` directory, which means it's available on the classpath for the application, so it's a simple matter to use the `getResourceAsStream()` method of the `Classloader` of the `DWikiContextListener` instance to get a stream on the properties file and use the Java standard `Properties` class to read it in.

Listing 5-18. *The Code of the* `DWikiContextListener` *Class*

```
package com.apress.dwrprojects.dwiki;

import java.io.InputStream;
import java.sql.Timestamp;
import java.util.HashMap;
import java.util.Properties;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class DWikiContextListener implements ServletContextListener {

  private static Log log = LogFactory.getLog(DWikiContextListener.class);

  public void contextInitialized (
    final ServletContextEvent inServletContextEvent) {

    log.trace("DWikiContextListener.contextInitialized() - Entry");

    try {

      InputStream is = this.getClass().getClassLoader().getResourceAsStream(
        "dwiki.properties");
      Properties props = new Properties();
      props.load(is);
      if (log.isDebugEnabled()) {
        log.debug("DWikiContextListener.contextInitialized() - props = " +
          props);
      }
```

```
      Config config = new Config();
      config.setDatabaseDriver((String)props.get("databaseDriver"));
      config.setDatabaseURI((String)props.get("databaseURI"));
      config.setDatabaseUsername((String)props.get("databaseUsername"));
      config.setDatabasePassword((String)props.get("databasePassword"));
      config.setEditLockTime((String)props.get("editLockTime"));
      if (log.isDebugEnabled()) {
        log.debug("DWikiContextListener.contextInitialized() - Config = " +
          config);
      }

      new DatabaseWorker().validateDatabase();

      new Freemarker().init(inServletContextEvent.getServletContext());

    } catch (Exception e) {
      log.error("DWikiContextListener.contextInitialized() - " +
        "Exception occurred during DWiki initialization.  " +
        "Application WILL NOT be available.  Error was: " + e);
    }

    log.trace("DWikiContextListener.contextInitialized() - Exit");

  } // End contextInitialized().

  public void contextDestroyed (
    final ServletContextEvent inServletContextEvent) {
  } // End contextDestroyed().

} // End class.
```

Once we have a `Properties` object populated with the data from the properties file, we just need to copy it over to our `Config` object. Recall that all the fields of that class are static, which means that after this is done, we have a place we can go to get those config values any time we want, for the life of the application, without having to reread the properties file.

The next task is to validate the database. To do this, we simply call the `validateDatabase()` method of the `DatabaseWorker` class. We'll be looking at what that method does soon, but for now just keep in mind that the result of that method is that the database will exist, with all necessary tables created, when it's done (or an exception will have occurred, essentially aborting startup of DWiki).

There's actually one more task that needs to be accomplished, and that's initializing FreeMarker. This is done to avoid unnecessary overhead when executing FreeMarker templates later. Again, we'll see precisely what goes on there very soon, but all that matters here is that it requires a call to the `init()` method of the `Freemarker` class. Note that this method requires the `ServletContext` to run.

Article.java

Like the `Config` class, the `Article` class is your everyday, run-of-the-mill simple javabean. You know, if Java just gave us structs like in C, we'd probably never see another javabean like this! But I digress. Figure 5-13 is the UML diagram for this class.

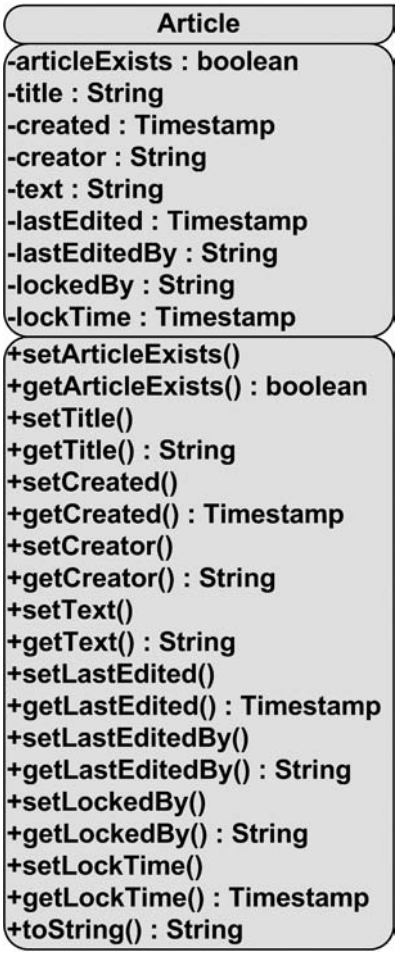


Figure 5-13. `Article` *class UML diagram*

The `title`, `created`, `creator`, and `text` fields are the four pieces of information that generically describe any article. The `text` field is literally the text of the article.

The `articleExists` field is a special flag that when `true` indicates that the article is not new. This is set to `false` when the user clicks an article link that leads to an article that does not yet exist. This is necessary information for the UI to work properly.

The `lastEdited` and `lastEditedBy` fields store the date/time the article was last modified and the username of the user who last modified it.

Finally, the `lockedBy` and `lockTime` are set when an article is being edited by someone. The `lockTime` is the date/time the lock was granted on the article and is used to calculate when a lock should be released.

After the fields are all the getters and setters for them, all their data types spelled out in the UML diagram, although I'd bet all of them are obvious based on only their names. And just like the `Config` class, the handy overridden `toString()` method is also present, making for easy debugging and logging.

ArticleComment.java

The `ArticleComment` class stores the details for one comment left for an article, and its UML diagram is shown in Figure 5-14.

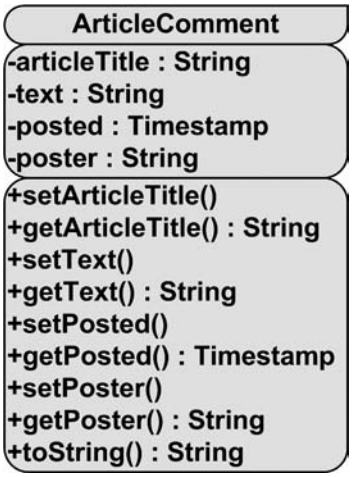


Figure 5-14. `ArticleComment` *class UML diagram*

As has become the pattern for these relatively simple classes, the code is not shown here. The fields of this class are `articleTitle`, which is essentially a foreign key to a particular article; `text`, which is the actual text of the comment; `posted`, which is the date/time the comment was left; and `poster`, which is, of course, the username of the user who left the comment. Recall that only registered users are allowed to leave comments. And once again, the ubiquitous custom `toString()` makes another appearance!

ArticleHistoryItem.java

The `ArticleHistoryItem` class, whose UML diagram you can see in Figure 5-15, is very nearly identical to the `ArticleComment` class in reality, and it describes a single modification event that occurred for an article. Once more, I've chosen not to show the code here owing to its extreme simplicity, but yet slightly page-filling girth.

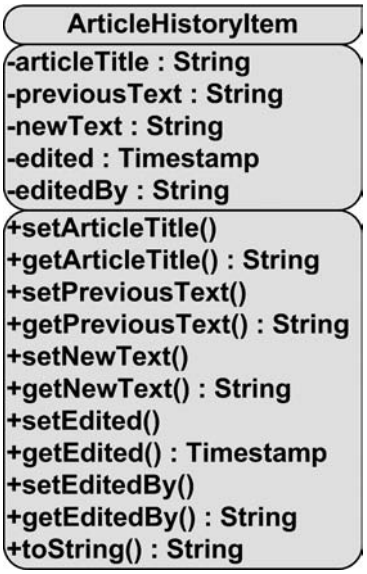


Figure 5-15. `ArticleHistoryItem` *class UML diagram*

As with `ArticleComment`, the `articleTitle` links a given history item to a particular article. The `previousText` field stores the entire contents of the article before an edit was made. The `newText` field stores the entire contents of the article after the edit was made (I mentioned earlier that this is a very inefficient and not very elegant way to do history, but it has the virtue of being very easy to implement, and really, it's not going to miss any changes, that's for sure!). The `edited` and `editedBy` fields serve the same purpose as `posted` and `poster` in the `ArticleComment` class.

And with that, we have only three more classes to look at, but I've saved the best, and lengthiest, for last. Let's start with the `Freemarker` class, which of the remaining three is probably the smallest and simplest.

Freemarker.java

Of course, you know by now that the FreeMarker templating engine is used to generate much of the output that is shown to the user in this application. Now we're going to see the class that actually does that work for us, and its UML diagram is shown in Figure 5-16.

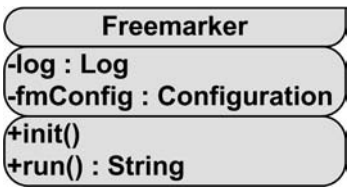


Figure 5-16. `Freemarker` *class UML diagam*

We've already seen how to use FreeMarker, so much of the code in Listing 5-19 will be familiar to you. The purpose of this class is to give the rest of the application code a common way to use FreeMarker, reducing duplicate code elsewhere. This class is a slightly modified version of the `Freemarker` class I donated to the DWR community, which you can find by digging through the DWR mailing list archives (this extension has not, at this time anyway, been accepted in any sort of official way, but it's out there, although you basically have it right here in front of your nose!).

Listing 5-19. *The* `Freemarker` *Class Code*

```
package com.apress.dwrprojects.dwiki;

import freemarker.cache.ClassTemplateLoader;
import freemarker.template.Configuration;
import freemarker.template.DefaultObjectWrapper;
import freemarker.template.Template;
import java.io.StringWriter;
import java.io.Writer;
import java.util.Map;
import javax.servlet.ServletContext;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Freemarker {

  private static Log log = LogFactory.getLog(Freemarker.class);

  private static Configuration fmConfig;

  public void init(ServletContext inServletContext) {

    if (fmConfig == null) {
      fmConfig = new Configuration();
      fmConfig.setObjectWrapper(new DefaultObjectWrapper());
      fmConfig.setServletContextForTemplateLoading(inServletContext,
        "templates");
    }

  } // End init().

  public String run(String inTemplate, Map inData) throws Exception {

    log.trace("Freemarker.run() - Entry");

    if (log.isDebugEnabled()) {
      log.debug(("Freemarker.run() - inTemplate = " + inTemplate));
```

```
      log.debug(("Freemarker.run() - inData = " + inData));
    }

    Template template = null;
    try {
      template = fmConfig.getTemplate(inTemplate);
    } catch (Exception e) {
      log.error("Freemarker.run() - Could not load Freemarker template " +
        inTemplate + "... is it in the classpath in the " +
        "expected location?  Is the template value passed in " +
        "fully-qualified? (Error: " + e);
      return "error";
    }

    Writer out = new StringWriter();
    template.process(inData, out);
    out.flush();
    if (log.isDebugEnabled()) {
      log.debug("Freemarker.run() - Generated output = " + out);
    }

    log.trace("Freemarker.run() - Exit");
    return out.toString();

  } // End run().


} // End class.
```

We first find the `init()` method, which you'll recall is called from the `DWikiContextListener`. You can see here that it's nothing but the FreeMarker configuration we saw earlier. Note that the `Configuration` object is stored as a static member. This is so we avoid the need to do this configuration with each template execution.

After that is the `run()` method, where a template is actually executed. To do so, all you need to do is pass in the name of the template to execute and the data model to feed to it. The code here then takes care of loading the template and executing it. The final result is a string is returned that is the result of the template execution, or the word "error" if the template couldn't be loaded. Also note that if any errors occur during template execution, the error itself will be part of the returned string. Good for debugging, bad for users!

DatabaseWorker.java

The `DatabaseWorker` class is a utility class that provides all our database functionality—from creating the database to executing queries against it, it's all in here. You can see the UML diagram for this class in Figure 5-17.

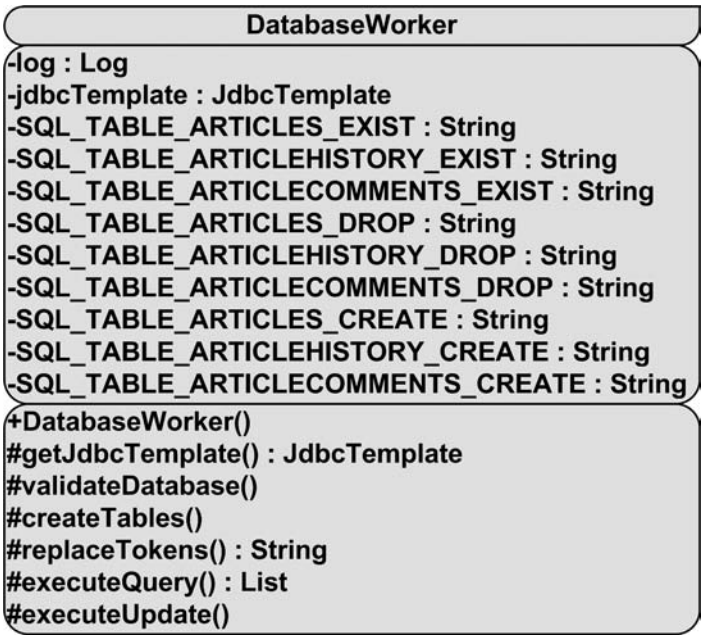


Figure 5-17. `DatabaseWorker` *class UML diagram*

Before we get into the code, we should discuss the database schema itself that DWiki uses. I've chosen to make it an extremely simple schema to avoid superfluous code here (this is already a pretty long chapter, no?). The schema is shown in Figure 5-18.

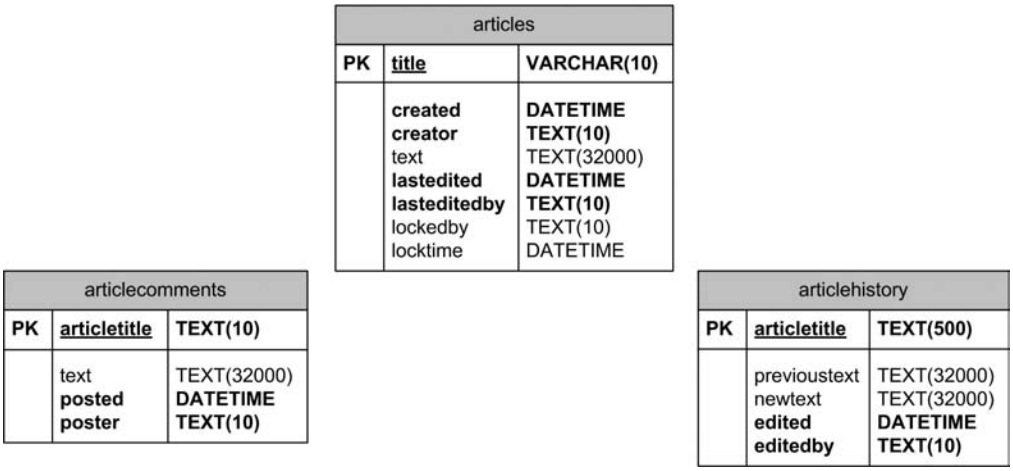


Figure 5-18. *The database schema of the application*

There are a whopping three tables in the database, so you won't be exclaiming "Well hello Mr. Fancy Pants"[3] any time soon! There are no constraints, triggers, referential integrity, or anything like that; it's just three loosely coupled tables. I say "loosely coupled" because they all do share the article's title as foreign keys to each other.

The `articles` table is obviously where articles themselves are stored. Note that there's a (roughly) 32KB limit on the size of an article. The fields in it should be pretty obvious, and they match the fields already shown in the `Article` class, so I won't go over them in detail here. Next is the `articlecomments` table, where again a given row has columns matching the fields in the `ArticleComment` class. Lastly, the `articlehistory` table matches the `ArticleHistoryItem` class from before. See, it all makes sense!

Because the `DatabaseWorker` class is a bit large, it's not shown in one big chunk here, but we'll look at its constituent parts now. The first few things you'll find are the Commons Logging `Log` instance and a `JdbcTemplate` field. This will be used later, but because multiple methods of the `DatabaseWorker` might be called during the processing of a request, it's more efficient to have only one instance of `JdbcTemplate` per request, and since `DatabaseWorker` will be instantiated once per request, making it a class member makes sense.

After that you'll find a series of nine constant strings, each being a SQL query. The first three, `SQL_TABLE_ARTICLES_EXIST`, `SQL_TABLE_ARTICLEHISTORY_EXIST`, and `SQL_TABLE_ARTICLECOMMENTS_EXIST`, are simple queries that just do a `select *` on a particular table (this is obviously not a great idea when the number of rows in the table grows larger, so you may want to modify those queries to something more reasonable—hint, hint!). The next three, `SQL_TABLE_ARTICLES_DROP`, `SQL_TABLE_ARTICLEHISTORY_DROP`, and `SQL_TABLE_ARTICLECOMMENTS_DROP`, are the queries to drop each of the tables. Lastly, `SQL_TABLE_ARTICLES_CREATE`, `SQL_TABLE_ARTICLEHISTORY_CREATE`, and `SQL_TABLE_ARTICLECOMMENTS_CREATE` create the three tables.

The constructor for the class does nothing but instantiate a `JdbcTemplate` object and store the reference to it. It does this by calling the `getJdbcTemplate()` method, which is this:

```
protected JdbcTemplate getJdbcTemplate() throws Exception {

  log.trace("DatabaseWorker.getJdbcTemplate() - Entry");

  DriverManagerDataSource dataSource = new DriverManagerDataSource();
  dataSource.setDriverClassName(Config.getDatabaseDriver());
  dataSource.setUrl(Config.getDatabaseURI());
  dataSource.setUsername(Config.getDatabaseUsername());
  dataSource.setPassword(Config.getDatabasePassword());

  log.trace("DatabaseWorker.getJdbcTemplate() - Exit");
  return new JdbcTemplate(dataSource);

} //End getJdbcTemplate().
```

3. This is a line spoken by the character Ash in the movie *Army of Darkness*. By the way, you can tell whether you're a true geek if when the old man tells Ash the magic words to make the Necronomicon safe ("Klaatu Barada Nikto"), you laugh your butt off and know exactly where they're from (the 1955 sci-fi classic *This Island Earth* . . . bonus points if you know the robot's name).

We already saw code like this when we initially discussed Spring JDBC, so it shouldn't be anything new. We're using the values stored in the `Config` object this time though, so if you wanted to connect to another RDBMS, it shouldn't require more than changing the properties file appropriately.

Next up is the `validateDatabase()` method that we saw called earlier from the `DWikiContextListener` class:

```
protected void validateDatabase() throws Exception {

  log.trace("DatabaseWorker.validateDatabase() - Entry");

  try {
    log.trace("DatabaseWorker.validateDatabase() - Checking tables");
    jdbcTemplate.queryForList(SQL_TABLE_ARTICLES_EXIST);
    jdbcTemplate.queryForList(SQL_TABLE_ARTICLEHISTORY_EXIST);
    jdbcTemplate.queryForList(SQL_TABLE_ARTICLECOMMENTS_EXIST);
  } catch (Exception e) {
    createTables();
  }

  log.trace("DatabaseWorker.validateDatabase() - Exit");

} // End validateDatabase().
```

So, here's the story: the application starts up, and `DWikiContextListener` fires. It calls `validateDatabase()` here, which then tries to query each of the tables. If an exception is thrown, we take that to mean the database hasn't been properly initialized (i.e., a table doesn't exist most likely), so we're at that point going to create all the tables (remember that Derby would have automatically created the database itself for us upon first access, so we're good to go there). The call to `createTables()` happens in the catch in `validateDatabase()`, and that method is this:

```
protected void createTables() throws Exception {

  log.trace("DatabaseWorker.createTables() - Entry");

  log.info("DatabaseWorker.createTables() - Dropping tables");
  try {
    jdbcTemplate.execute(SQL_TABLE_ARTICLES_DROP);
  } catch (Exception e) { }
  try {
    jdbcTemplate.execute(SQL_TABLE_ARTICLEHISTORY_DROP);
  } catch (Exception e) { }
  try {
    jdbcTemplate.execute(SQL_TABLE_ARTICLECOMMENTS_DROP);
  } catch (Exception e) { }
```

```
    log.info("DatabaseWorker.createTables() - Creating tables");
    jdbcTemplate.execute(SQL_TABLE_ARTICLES_CREATE);
    jdbcTemplate.execute(SQL_TABLE_ARTICLEHISTORY_CREATE);
    jdbcTemplate.execute(SQL_TABLE_ARTICLECOMMENTS_CREATE);

    log.trace("DatabaseWorker.createTables() - Entry");

  } // End createTables().
```

Here, it's a simple matter of executing first the drop SQL statements for each table (to account for the case where perhaps one table exists but another doesn't), and then executing the table create SQL statements. Note that the drop executions are wrapped in `try...catch` to deal with dropping a table that doesn't exist (which will always happen if the database was created fresh).

Following `createTables()` is a little utility method used internally by `DatabaseWorker` called `replaceTokens()`. What happens is that when queries are passed in to either the `executeQuery()` or `executeUpdated()` methods (which we'll see after this), `replaceTokens()` is called, and its job, as its name implies, is to replace any tokens found in the SQL string with data passed in as a `Map`. So for instance, if we had this query:

```
select * from ${tableName}
```

and we passed in a `Map` with a single `String` named `tableName` with a value of `articles`, then `replaceTokens()` would result in the following being the final query that is actually executed:

```
select * from articles
```

This isn't anything unusual, of course, but it comes in handy for sure. The `replaceTokens()` method by the way is this:

```
  protected String replaceTokens(final String inString, final Map inVals) {

    log.trace("DatabaseWorker.replaceTokens() - Entry");

    if (log.isDebugEnabled()) {
      log.debug("DatabaseWorker.replaceTokens() - inString = " + inString);
      log.debug("DatabaseWorker.replaceTokens() - inVals = " + inVals);
    }

    String retString = inString;
    for (Iterator it = inVals.keySet().iterator(); it.hasNext();) {
      String nextToken = (String)it.next();
      String nextValue =
        StringEscapeUtils.escapeSql((String)inVals.get(nextToken));
      retString = StringUtils.replace(retString, "${" + nextToken + "}",
        nextValue);
    }
```

```
    if (log.isDebugEnabled()) {
      log.debug("DatabaseWorker.replaceTokens() - retString = " + retString);
    }

    log.trace("DatabaseWorker.replaceTokens() - Exit");
    return retString;

  } // End replaceTokens().
```

Note that this method (and indeed all the methods in `DataWorker`) is protected so that you can override it with different implementations if you wish. This might be especially valuable for `replaceTokens()`, for which a more efficient method may well exist (although using Commons Lang's `StringUtils.replace()` method works very nicely). Also note here that the `StringEscapeUtils.escapeSql()` method is used on each token value as it is inserted, so that the SQL statement doesn't get broken by bad characters (primarily this means escaping the ' character into '').

Now we come to `executeQuery()`, which obviously executes a SQL query and returns a `List` of results:

```
  protected List executeQuery(final String inSQL, final Map inVals)
    throws Exception {

    log.trace("DatabaseWorker.executeQuery() - Entry");

    if (log.isDebugEnabled()) {
      log.debug("DatabaseWorker.executeQuery() - inSQL = " + inSQL);
      log.debug("DatabaseWorker.executeQuery() - inVals = " + inVals);
    }

    String sql = replaceTokens(inSQL, inVals);

    List data = jdbcTemplate.queryForList(sql);
    if (log.isDebugEnabled()) {
      log.debug("DatabaseWorker.executeQuery() - Returned List = " + data);
    }

    log.trace("DatabaseWorker.executeQuery() - Exit");
    return data;

  } // End executeQuery().
```

Once some logging is done, and tokens are replaced in the incoming SQL, it's just a call to the `queryForList()` method of the Spring `JdbcTemplate` object, and that's that.

The `executeUpdate()` method is next, and it's virtually identical to `executeQuery()`, so it won't be shown here. The only notable difference is that instead of calling `queryForList()`, the call is to `update()` of the `JdbcTemplate` object.

ArticleDAO.java

To this point, we've seen the client side of DWiki. We've seen the server side, how FreeMarker is used to generate output, how various Value Objects, or VOs, are used to pass data around and between the UI and the server. We've seen how the `DatabaseWorker` gives us a level of abstraction away from the underlying database and a nice, clear API to use to work with our data. We've seen how Spring's JDBC support makes the code in `DatabaseWorker` very minimal.

Now, at long last, we come to the single class that is truly the heart and soul of DWiki, certainly as far as the server side of things goes. We've seen this class called numerous times from the client, so we have a good idea what each method in it does already, but now we'll see the implementation details involved. That class is, of course, the `ArticleDAO` class, and its UML class diagram is shown in Figure 5-19.

Figure 5-19. `ArticleDAO` *class UML diagram*

`ArticleDAO` is another pretty large piece of code, coming in at just over 800 lines of source code, so we'll just tackle each piece of it as needed here, beginning with the class members.

We again have a `Log` instance to use throughout. We then have a batch of 11 constants that are SQL queries, just like in `DatabaseWorker`. These fields are summarized in Table 5-2. Following these constants is a reference to an instance of a `DatabaseWorker` object, and this instance is created in the constructor of this class (that's literally all the constructor does, hence it isn't shown here). You should note that this means that each request from the client will result in DWR creating a new instance of `ArticleDAO`, which means a new instance of `DatabaseWorker`, which also means a new instance of `JdbcTemplate` because, remember, `DatabaseWorker` creates an instance in its constructor. So, for each request, we are guaranteed that only a single database connection is used, which is a good design because it ensures, assuming you have some connection pooling implemented properly, that you'll have a nice, efficient use of database connections, which is always a concern.

Table 5-2. *The Constants (All Strings) Defined in* `ArticleDAO`

Field Name	Description
`SQL_ADD_ARTICLE`	SQL for adding an article
`SQL_GET_ARTICLE`	SQL for getting a single article
`SQL_ADD_ARTICLE_HISTORY`	SQL for adding a history item for an article
`SQL_GET_ARTICLE_HISTORY`	SQL for getting a single article's history
`SQL_ADD_ARTICLE_COMMENT`	SQL for adding a comment for an article
`SQL_GET_ARTICLE_COMMENTS`	SQL for getting a single article's comments
`SQL_GET_ARTICLES`	SQL for getting a list of all articles
`SQL_UPDATE_ARTICLE`	SQL for updating an article
`SQL_FIND_ARTICLE_BY_TITLE`	SQL for searching for articles by title
`SQL_FIND_ARTICLE_BY_TEXT`	SQL for searching for articles by article text
`SQL_FIND_ARTICLE_BY_BOTH`	SQL for searching for articles by both article title and text

addArticle() Method

The first method we come to after all the class members and the constructor is `addArticle()`:

```
public Article addArticle(final String inArticleTitle) throws Exception {

  log.trace("ArticleDAO.addArticle() - Entry");

  try {

    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.addArticle() - inArticleTitle = " + inArticleTitle);
    }
```

```
      HashMap<String, Object> vals = new HashMap<String, Object>();
      vals.put("articleTitle", inArticleTitle);
      vals.put("text", new Freemarker().run("newArticle.ftl", new HashMap()));

      databaseWorker.executeUpdate(SQL_ADD_ARTICLE, vals);
      Article article = getArticle(inArticleTitle, null, null, true);

      log.trace("ArticleDAO.addArticle() - Exit");
      return article;

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End addArticle().
```

Adding an article is a simple matter of putting the title of the article and the text of the article into a `HashMap` and handing that, plus the appropriate SQL statement, off to `DatabaseWorker`'s `executeUpdate()` method. The interesting thing is that, as you will recall, when an article is initially created, it is basically just some placeholder content. Rather than hard-code that content in this code, I used a FreeMarker template to generate it. That template doesn't require any data to be inserted, hence the empty `HashMap` passed into the `run()` method of the `Freemarker` class.

Once the article is added, a call is immediately done to `getArticle()` because the return value from `addArticle()` is actually an `Article` object, so rather than manually construct that object here (and we wouldn't have all the details to do that anyway), we read the newly created record from the database to save us the trouble.

getArticle() Method

The `getArticle()` method is, conveniently, next in order in the code. It's actually a pretty substantial chunk of code, so let's break it down. After some initial logging, we have a couple of "special" situations to deal with:

```
    if (inArticleTitle.equals("AllArticlesIndex")) {
      article = getAllArticles();
      doLinkExpansion = false;
    } else if (inArticleTitle.equals("DWikiHelp")) {
      article = getStaticArticle("DWikiHelp", new HashMap());
    } else if (inArticleTitle.equals("Search")) {
      article = getStaticArticle("Search", new HashMap());
      doLinkExpansion = false;
    } else if (inArticleTitle.equals("SearchResults")) {
      article = search(inExtraData1, inExtraData2);
      doLinkExpansion = false;
    } else {
      ...
```

These are what we'll call our "system" articles, and they require some special processing, hence the branching logic. For the list of all articles on the wiki, we make a call to `getAllArticles()`, which will return an `Article` object where the text of the article is the actual list. Note the `doLinkExpansion = false;` line. After an article is retrieved, the text of it is scanned for words that begin with a capital letter, and these are "expanded" into a hypertext link for the user to click to go to that article (or create a new one). However, in some of these system article cases, we don't want that link expansion to occur, so setting this flag indicates to the downstream code to not expand links for this article.

For the help article, it's just a call to the `getStaticArticle()` method, which is used to return a truly static, but noneditable, article. This also goes for the search view, which, even though it doesn't seem like it, in fact **is** an article as far as DWiki is concerned. The last branch is for search results. When the user enters search criteria, you'll recall that the client calls `getArticle()`, with "SearchResults" as the requested article title. Well, we clearly have to actually do a search in this case, hence the call to `search()` here. The search is performed, and the results are returned as an `Article` object with the actual search results as the text. Once again, virtually everything is an article, which makes the code fairly generic. Note too how the two extra data elements that we saw earlier in the client DWR calls is used here as our search criteria (the first is the text to search for, the second is what to search).

Now, let's assume we're not dealing with one of those special cases. In that case, the following code gets executed:

```
HashMap<String, String> vals = new HashMap<String, String>();
vals.put("articleTitle", inArticleTitle);

List articles = databaseWorker.executeQuery(SQL_GET_ARTICLE, vals);

article = new Article();
if (articles.size() == 0) {
  log.debug("Article NOT found, creating default");
  doLinkExpansion = false;
  article.setArticleExists(false);
  article.setTitle("inArticleTitle");
  article.setCreated(new Timestamp(new java.util.Date().getTime()));
  article.setCreator("SYSTEM");
  Map<String, Object> tokens = new HashMap<String, Object>();
  tokens.put("articleTitle", inArticleTitle);
  article.setText(new Freemarker().run("nonexistentArticle.ftl",
    tokens));
} else {
  log.debug("ArticleDAO.getArticle() - Article found");
  Map m = (Map)articles.get(0);
  article.setTitle((String)m.get("TITLE"));
  article.setCreated((Timestamp)m.get("CREATED"));
  article.setCreator((String)m.get("CREATOR"));
  article.setText((String)m.get("TEXT"));
  article.setLastEdited((Timestamp)m.get("LASTEDITED"));
```

```
        article.setLastEditedBy((String)m.get("LASTEDITEDBY"));
        article.setLockedBy((String)m.get("LOCKEDBY"));
        article.setLockTime((Timestamp)m.get("LOCKTIME"));
      }
```

A call is made to `executeQuery()` to get the list of articles with the specified title. We then check to see whether no data was returned, which means the user must have clicked a link to an article that doesn't yet exist. So, we populate the `Article` instance with some "starter" data so to speak, including the output of the `nonexistentArticle.ftl` FreeMarker template. If the article is found (and here we're assuming only a single row of data was returned, although that's technically not necessarily true because there's no unique constraint on the articles table's `articleTitle` field, which there really should be), the `Article` instance is populated with the data returned from the database.

At the end of this, we have an `Article` object, one way or another, whether it was one of the special cases, an existing article, or a newly created article. We then call the `expandLinks()` method, if and only if the `doLinkExpansion` flag is set to `true`. We set the text of the article to what that method returns, and return the `Article` object, and we've supplied the client with the requested article!

expandLinks() Method

Speaking of the `expandLinks()` method, let's see what that's all about now, shall we?

```
protected String expandLinks(final String inText) {

  log.trace("ArticleDAO.expandLinks() - Entry");

  StringBuffer sb = new StringBuffer(inText.length() + 1024);

  StringTokenizer st = new StringTokenizer(inText, " \t\n\r\f\b\0", true);
  while (st.hasMoreTokens()) {
    String nextWord = st.nextToken();
    if (nextWord.length() > 1) {
      if (nextWord.charAt(0) == '~') {
        nextWord = nextWord.substring(1);
      } else {
        if (Character.isUpperCase(nextWord.charAt(0))) {
          nextWord = new StringBuffer(512).append(
            "<a href=\"javascript:void(0);\" ").append(
            "onClick=\"DWiki.getArticle('").append(nextWord).append(
            "');\">").append(nextWord).append("</a>").toString();
        }
      }
    }
    sb.append(nextWord);
  }
```

```
    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.expandLinks() - sb.toString() = " + sb.toString());
    }

    log.trace("ArticleDAO.expandLinks() - Exit");
    return sb.toString();

  } // End expandLinks().
```

This is another one of those cases where I don't doubt there's a more efficient algorithm, so I made this method protected so it can be overridden easily (otherwise, it probably would have made sense to be private since it's only used as a utility function in this class). The actual link expansion mechanism is simple: break up the text of the article, as passed in as the `inText` parameter, breaking on all whitespace. Note that breaking only on spaces, which you might think would work since you're concerned with each individual word, doesn't quite work because you get tokens where you don't expect them, or don't get tokens where you should, and the expansion doesn't work out. Also note that whitespace is included in each token, so that the whitespace is ultimately kept when the text we're constructing here is being built up. Then, for each token (which nominally actually *is* a word), you first make sure that the token's length is greater than one (because a capital letter "I" shouldn't be an article link, for instance). Then, you check to see whether the first character is a tilde, which is our indicator to say that the following word, even though it normally would be, should not be an article link. If it is, we set the `nextWord` variable to the token, starting with the **next** character, so for instance, `~Dog` will cause `nextWord` to be `Dog`, which will then be inserted into the text.

Now, if we figure out that the word should be an article link, `nextWord` is set to the markup for such a link, which you can see. Ultimately, whatever the outcome of all that logic was, the value of `nextWord` is appended to our `StringBuffer` that is building up (which is why we needed to keep whitespace, by the way; otherwise, "Who's more foolish, the fool or the fool who follows him?" would wind up being "Who'smorefoolishthefoolorthefoolwhofollowshim?"), the net result of which is that the `StringBuffer` winds up containing the text of the article, with all article links expanded into actual clickable links, and that's what the method ultimately returns.

getAllArticles() Method

Following the `expandLinks()` method is `getAllArticles()`, which you'll recall is called from `getArticle()` when the list of all articles is requested.

```
  private Article getAllArticles() throws Exception {

    log.trace("ArticleDAO.getAllArticles() - Entry");

    try {

      List articles = databaseWorker.executeQuery(
        SQL_GET_ARTICLES, new HashMap());
      if (log.isDebugEnabled()) {
        log.debug("ArticleDAO.getAllArticles() - articles = " + articles);
      }
```

```
      Map<String, Object> tokens = new HashMap<String, Object>();
      tokens.put("articles", articles);
      Article article = getStaticArticle("AllArticlesIndex", tokens);

      log.trace("ArticleDAO.getAllArticles() - Exit");
      return article;

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End getAllArticles().
```

Only two major steps are required. First, the `SQL_GET_ARTICLES` query is executed, which gets us a list of all articles, but only the title, creator, and date/time created for each. Note that this execution gets us a `List` of `Maps` back, because that's what the `queryForList()` method of the `JdbcTemplate` class returns. We simply add this `List` to a `Map`, which is now the data model for the FreeMarker template that will generate the output.

getStaticArticle() Method

In order to execute the template, a call is next made to `getStaticArticle()`, which we'll look at now:

```
  private Article getStaticArticle(final String inTitle, final Map inTokens)
    throws Exception {

    log.trace("ArticleDAO.getStaticArticle() - Entry");

    try {

      Article article = new Article();
      article.setTitle(inTitle);
      article.setCreator("SYSTEM");
      article.setCreated(new Timestamp(new java.util.Date().getTime()));
      article.setText(new Freemarker().run(inTitle + ".ftl", inTokens));

      log.trace("ArticleDAO.getStaticArticle() - Exit");
      return article;

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End getStaticArticle().
```

This method simply creates an `Article` object and populates it. The article's title is passed in, and the `creator` and `created` fields are set to static values (static in the sense that they aren't fed to the method). Then, it's a simple matter to set the text field of the `Article` object to the output of the FreeMarker template, which is determined by using the template name passed in (`inTitle`). The `Article` instance is then returned, which, going back to the `getAllArticles()` method, means that it can then return the `Article` instance to the original caller, giving it the list of all articles it asked for. This `getStaticArticle()` method makes an appearance in a number of other methods, as we already saw back in `getArticle()`.

getArticleHistory() Method

We'll jump around a little and next look at the `getArticleHistory()` method:

```
public String getArticleHistory(final String inArticleTitle)
  throws Exception {

  log.trace("ArticleDAO.getArticleHistory() - Entry");

  try {

    // Log incoming parameters.
    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.getArticleHistory() - inArticleTitle = " +
        inArticleTitle);
    }

    // Construct Map of parameters for SQL query.
    HashMap<String, String> vals = new HashMap<String, String>();
    vals.put("articleTitle", inArticleTitle);

    // Get list of all history for the article.
    List historyItems = databaseWorker.executeQuery(
      SQL_GET_ARTICLE_HISTORY, vals);
    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.getArticleHistory() - historyItems = " +
        historyItems);
    }

    // Construct Map of parameters for Freemarker template.
    Map<String, Object> tokens = new HashMap<String, Object>();
    tokens.put("historyItems", historyItems);
    String output = new Freemarker().run("ArticleHistory.ftl", tokens);
    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.getArticleHistory() - output = " +
        output);
    }
```

```
      log.trace("ArticleDAO.getArticleHistory() - Exit");
      return output;

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End getArticleHistory().
```

Retrieving the history records for an article amounts to not much more than executing the `SQL_GET_ARTICLE_HISTORY` query, inserting the article's title into it, and taking the `List` returned and feeding it to the `ArticleHistory.ftl` template. It should look pretty mundane to you at this point!

getArticleComments() Method

Very similar to `getArticleHistory()` is `getArticleComments()`. They are so similar, in fact, that I haven't shown `getArticleComments()` here. The only real differences are that obviously `getArticleComments()` deals with `ArticleComment` objects rather than `ArticleHistoryItem` objects, and that when creating the data model to feed to the FreeMarker template, we add a flag that indicates whether the user is a registered DWiki user or not. To do this, we make a call to the `isUserInRole()` method of the `HttpServletRequest` object, which you'll notice is a parameter to the method. Recall that DWR will automagically provide that object when it sees it as a method parameter; that's why you didn't see any indication of this parameter in the client-side call to this method.

addArticleHistory() Method

Now that we've seen how to get article history and comments, let's see how to add both. I'll just be showing the `addArticleHistory()` method here because `addComment()` is pretty much the same thing, just like the get methods were pretty much the same.

```
  private void addArticleHistory(final ArticleHistoryItem inArticleHistoryItem)
    throws Exception {

    log.trace("ArticleDAO.addArticleHistory() - Entry");

    try {

      if (log.isDebugEnabled()) {
        log.debug("ArticleDAO.addArticleHistory() - inArticleHistoryItem = " +
          inArticleHistoryItem);
      }

      HashMap<String, String> vals = new HashMap<String, String>();
      vals.put("articleTitle", inArticleHistoryItem.getArticleTitle());
```

```
      vals.put("previousText", inArticleHistoryItem.getPreviousText());
      vals.put("newText", inArticleHistoryItem.getNewText());
      vals.put("editedBy", inArticleHistoryItem.getEditedBy());

      databaseWorker.executeUpdate(SQL_ADD_ARTICLE_HISTORY, vals);

      log.trace("ArticleDAO.addArticleHistory() - Exit");

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End addArticleHistory().
```

It doesn't take a whole lot to add a history item for an article. An `ArticleHistoryItem` object is passed in, which contains all the necessary information. Then it's just a simple matter of taking that information and putting it into a `HashMap` to pass in with the `SQL_ADD_ARTICLE_HISTORY SQL` statement; `DatabaseWorker` does its job of inserting the data into the SQL, and we're off to the races.

addComment() Method

As I mentioned, `addComment()` is more or less the same, the one important difference being that we need the name of the user adding the comment, and to get that we make a call to `getUserPrincipal().getName()` of the `HttpServletRequest` object, which is the final parameter to the method and so is provided once again by DWR.

updateArticle() Method

We've jumped around a bit here and skipped the `updateArticle()` method, the one that's called when a user edits an article and clicks the Save button, so let's go back to that now:

```
  public void updateArticle(final Article inArticle,
    final boolean inWriteHistory, final HttpServletRequest inRequest)
    throws Exception {

    log.trace("articleDAO.updateArticle() - Entry");

    try {

      if (log.isDebugEnabled()) {
        log.debug("ArticleDAO.updateArticle() - inArticle = " + inArticle);
      }

      String originalText =
        getArticle(inArticle.getTitle(), null, null, false).getText();
```

```
      HashMap<String, String> vals = new HashMap<String, String>();
      vals.put("text", inArticle.getText());
      vals.put("title", inArticle.getTitle());
      vals.put("lastEditedBy", inRequest.getUserPrincipal().getName());
      if (inArticle.getLockedBy() == null) {
        vals.put("lockedBy", "");
        vals.put("lockTime",
          new Timestamp(new java.util.Date().getTime()).toString());
      } else {
        vals.put("lockedBy", inArticle.getLockedBy());
        vals.put("lockTime", inArticle.getLockTime().toString());
      }

      databaseWorker.executeUpdate(SQL_UPDATE_ARTICLE, vals);

      if (inWriteHistory == true) {
        ArticleHistoryItem historyItem = new ArticleHistoryItem();
        historyItem.setArticleTitle(inArticle.getTitle());
        historyItem.setPreviousText(originalText);
        historyItem.setNewText(inArticle.getText());
        historyItem.setEditedBy(inRequest.getUserPrincipal().getName());
        addArticleHistory(historyItem);
      }

      log.trace("ArticleDAO.updateArticle() - Exit");

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End updateArticle().
```

So, this is a little more interesting, no? The first thing that's done is to retrieve the original text of the article from the database. As you'll recall, I chose to implement a very simplistic history mechanism whereby we record the text of the article before and after the edit. The problem here is that we don't actually have the original text any more ready to be written, but it's still in the database, so a call is made to `getArticle()`, and the text of the returned `Article` object is grabbed.

Next, the incoming `Article` object containing the new text needs to be saved, and that's a simple matter of taking the data from it and putting it in a `HashMap`, then calling `DatabaseWorker.executeUpdate()`, passing it the `SQL_UPDATE_ARTICLE` statement and the `HashMap`.

The next task is to write the history record so we know who made the edit and when. To do this, an `ArticleHistoryItem` is created and populated, and a call to `addArticleHistory()`, which we already saw, is made.

Note that both saving the article and adding the history record require the username, so again we see the `HttpServletRequest` object as a method parameter and a call to `getUserPrincipal().getName()` being used.

Oh yeah, one bit of trickiness I skipped over here: did you notice the `if` check to see whether the `lockedBy` field has a value? And did you notice the check of `inWriteHistory` around the history record creation? Those are necessary because when a user requests an article be locked so he or she can edit it, we need to record the lock information on the record for the article in the database. However, in that situation, we don't want to create a history record. So, `updateArticle()` will be called when a user is given an edit lock, and the `inWriteHistory` flag will be `false`, which will cause the history record to not be written. Likewise, when an article is being locked, the incoming `Article` object will contain values in the `lockedBy` and `lockTime` fields; otherwise it won't, and when it doesn't, which is the case when the updates to the article are actually being saved, we need to make sure those two fields get cleared in the database, so the `if` statement branches to clear them.

lockArticleForEditing() Method

Speaking of locking an article, that's accomplished by a call to `lockArticleForEditing()`, which is a logical segue from `updateArticle()` since they play together.

```
public String lockArticleForEditing(final String inArticleTitle,
  final HttpServletRequest inRequest) throws Exception {

  log.trace("ArticleDAO.lockArticleForEditing() - Entry");

  try {

    Article article = getArticle(inArticleTitle, null, null, false);
    String retVal = "";

    if (article.getLockedBy() == null ||
      article.getLockedBy().equals(inRequest.getUserPrincipal().getName())) {


      log.debug("Article NOT locked, acquiring lock on behalf of user");
      article.setLockedBy(inRequest.getUserPrincipal().getName());
      article.setLockTime(new Timestamp(new java.util.Date().getTime()));
      updateArticle(article, false, inRequest);
      retVal = "ok=" + Config.getEditLockTime() + "=" + article.getText();

    } else {


      log.debug("Article WAS locked, checking for lock expiration");
      int editLockTime = Integer.parseInt(Config.getEditLockTime());
      long numMillis = 1000 * editLockTime;
```

```
        long lockDiff = new Timestamp(new java.util.Date().getTime()).getTime() -
          article.getLockTime().getTime();
        if (lockDiff > numMillis) {
          log.debug("Lock has expired, acquiring lock on behalf of new user");
          article.setLockedBy(inRequest.getUserPrincipal().getName());
          article.setLockTime(new Timestamp(new java.util.Date().getTime()));
          updateArticle(article, false, inRequest);
          retVal = "ok=" + Config.getEditLockTime();
        } else {
          log.debug("Lock has NOT expired, informing user they cannot edit");
          retVal = "lockedBy=" + article.getLockedBy() + "=" +
            (editLockTime - (lockDiff / 1000));
        }

      }

      log.trace("ArticleDAO.lockArticleForEditing() - Exit");
      return retVal;

    } catch (Exception e) {
      e.printStackTrace();
      throw e;
    }

  } // End lockArticleForEditing().
```

This is probably the meatiest method in the whole `ArticleDAO` class. First, we call the `getArticle()` method to get an `Article` object for the specified article (to lock an article only the title is needed). Once we have that `Article` object, we check to see whether it's already locked. If it isn't, or if it's locked by the user currently requesting the lock, it can be locked. To do so, we set the `lockedBy` field to the name of the user making the request, using `HttpServletRequest`'s `getUserPrincipal().getName()` method (the request object is again a method parameter autopopulated by DWR). Then it's just a call to `updateArticle()`, this time with the `inWriteHistory` flag set to `false`, for the reasons described previously.

Now, let's follow the case of the article already being locked. Two things can happen here. First, what if the lock hasn't yet expired? That's the easy case (it's the `else` branch of the `if`): all we need to do is return a string in the form `lockedBy=xxx=yyy`, where `xxx` is the name of the user who has the article locked and `yyy` is the amount of time remaining on the lock. The UI will recognize this string and pop an `alert()` message to the user telling that user he or she can't have the edit lock.

If the lock has expired though, that means the new user can have the lock on the article (and as we've seen previously, it means the user that currently has the lock won't be able to save his or her changes . . . tough luck, bub!). When this situation occurs, the same flow is followed for obtaining a lock in the first place, and the string `ok=xxx` is returned, where `xxx` is the amount of time the lock will last for.

search() Method

Finally, at long last, we've reached the last bit of code in the entire DWiki application that we've yet to explore, and that's the `search()` method for performing an article search on behalf of the user.

```
private Article search(final String inSearchText, final String inSearchWhat)
  throws Exception {

  log.trace("ArticleDAO.search() - Entry");

  try {

    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.search() - inSearchText = " + inSearchText);
      log.debug("ArticleDAO.search() - inSearchWhat = " + inSearchWhat);
    }

    HashMap<String, Object> vals = new HashMap<String, Object>();
    vals.put("searchText", inSearchText);

    String sql = null;
    if (inSearchWhat.equals("title")) {
      sql = SQL_FIND_ARTICLE_BY_TITLE;
    } else if (inSearchWhat.equals("text")) {
      sql = SQL_FIND_ARTICLE_BY_TEXT;
    } else if (inSearchWhat.equals("both")) {
      sql = SQL_FIND_ARTICLE_BY_BOTH;
    }

    List articles = databaseWorker.executeQuery(sql, vals);
    if (log.isDebugEnabled()) {
      log.debug("ArticleDAO.search() - articles = " + articles);
    }

    Map<String, Object> tokens = new HashMap<String, Object>();
    tokens.put("articles", articles);
    Article article = getStaticArticle("SearchResults", tokens);

    log.trace("ArticleDAO.search() - Exit");
    return article;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

} // End search().
```

As it turns out, searching for an article is pretty darned easy. The parameters we get are the text to search for (`inSearchText`) and what to search (`inSearchWhat`, which is one of the values `"title"`, `"text"`, or `"both"`). We start by creating a `HashMap` and putting `inSearchText` into it under the key `searchText`. Then, we branch on `inSearchWhat`, and select a different SQL statement constant based on it. Note that all three use a `like` clause to do the search, and each converts the text being searched in the record, as well as the text being searched for, to uppercase, so that we get a case-insensitive search in all cases. Then, we execute the query, and pass the resultant `List` of article records, if any, to the `getStaticArticle()` method. This `List`, which holds the replacement tokens that will be used to generate the output in `getStaticArticle()`, is put into a `Map`. We request the `SearchResults` static article, which executes the `SearchResults.ftl` FreeMarker template, which uses the `List` of article records to generate the output.

And with that, our exploration of DWiki is done! I hope you've found this (somewhat lengthy) journey to be fun, exciting, and worthy of your time. I mean, I realize it's not Jessica Alba in a bikini (or Brad Pitt in a speedo for the ladies), but it was enjoyable, wasn't it?

Suggested Exercises

Although DWiki is very useful as-is, there are quite a few features I left out so you'd have a good chance to practice your DWR fu. Here are just a few suggestions, most of which shouldn't be too tough to implement at all.

- Allow users to self-register. This is not nearly as easy as it sounds! You can't modify the Tomcat user list on the fly, so you'll need to work out a whole other login mechanism. But keep in mind that you'll still want to use the J2EE security mechanism too, so the trick is to somehow validate the user against the database (where you would have created a users table and written a record out to when they registered) and only then log in via container-managed security. Of course, you'll want this to be transparent to the user. Like I said, this one is a significant challenge, but I think you're up to it!

- Add some administrative functions. This was actually in my original plans for the application, but time constraints came into play, so I had to drop them. Here's what I planned: there would be an Admin link on the left. When clicked, you'd be able to delete articles as well as users (assuming you implemented the first suggestion). This would require, aside from the obvious UI changes, delete methods in both the `ArticleDAO` and `UserDAO` (which I'd presume you added with the first suggestion); a method to list all users in the `UserDAO`; a new security role, `dwiki_admin`; and the corresponding configuration to lock down those functions in `dwr.xml`. You can go ahead and implement this as described, or add your own functionality as you see fit.

- Add the ability to attach files to an article. I suggest exploring the Commons FileUpload component for this.

- Implement a more robust history system. I took probably the easiest approach here, but something better would be nice. I'm thinking something like a rudimentary diff display, at least a list of lines that were changed with the original and new line.

- Add a Recent Changes link on the left that shows the last, say, ten articles to be edited.

- Enhance the database structure to allow for larger articles. I would suggest a fourth table that stored 32KB chunks of the article, as many as there are, and then you just stitch them all together to form the article. You may have another approach though—it's all good!
- Add favorites for users. Allow users to mark an article as a favorite, and give them a link on the left to display a list of all their favorites. This would obviously require changes to the database structure, `UserDAO`, UI, and probably `ArticleDAO`.

Summary

The DWiki project, I believe, is a great example of DWR usage and Ajax in general. You saw in this chapter how you can leverage container-managed J2EE security to lock down methods that you remote with DWR. You saw how you can use the FreeMarker templating engine, in conjunction with DWR, to generate output in a flexible, maintainable fashion. You also saw some of Spring JDBC and the Derby database package in action. Best of all, you have a handy little wiki application that you could quite conceivably deploy for real and make good use of to collaborate with people. If you implemented any of the suggested exercises, you extended it even further and learned more about DWR in the process.

In the next chapter, we'll implement another extremely useful application, namely a file manager, and we'll see how DWR can make such a thing work quite a bit like the file manager you are probably used to in your native operating system.

CHAPTER 6

Remotely Managing Your Files: DWR File Manager

Imagine you have a nice, fast, fat pipe to your house (get your mind out of the gutter **right now!!**). You would love to be able to set yourself up a server that you could access from anywhere on which you could keep your files. Further imagine that you'd like to have something akin to Windows Explorer available to you, a relatively basic file manager that would allow you to mess around with those files on that server at will.

Well, with the help of DWR, some nifty UI components, and some very handy utility libraries, that's exactly what we'll be building in this chapter! We're going to cover the basics that you'd expect to find in a file manager, and perhaps one or two things beyond that. We'll use DWR to connect the user interface with the server-based components as we've seen before, and we'll also put some decent security on top of this, given how dangerous such an application could obviously be. The best part is, we'll pull all this off with a surprisingly sparse amount of code.

For short, let's call this project Fileman . . . just so my fingers don't get tired typing the rest of this chapter! So, with that small detail out of the way, let's jump right in and begin by discussing exactly what this application is going to do, and then get right down to some code.

Application Requirements and Goals

If you are using Windows right now, pop open Windows Explorer, and by and large, you'll already know what we're going to build here. Windows Explorer is a generally pretty basic file manager, but one that gets the job done, to a first approximation, fairly well. There are certainly better file managers out there (see the sidebar "A Bevy of File Managers" for a few if you're interested), but for something that comes with your operating system, it's not too bad either.

So, in enumerating what we're trying to accomplish with this project, we don't have to look much further than Windows Explorer itself.

- We want to provide the ability to see a tree view of the directory structure of the server the application is running on. We should be able to expand and contract branches of the tree at will. This tree should have as its first-level nodes the root file systems on the server. On Windows systems, this means drive letters; on *nix systems, it means mount points. In addition, we should always have the ability to go to the parent of the directory we're in and the ability to refresh the current directory.

- We should have the basic operations of copying files and directories, moving files and directories (both of these are basically two-phase operations, first where we mark either the file or directory as being copied or cut, and then the second phase of pasting it somewhere), renaming files and directories, and creating new files and directories.
- We should be able to edit files in a simplistic text editor. Obviously, this won't help for editing complex file types like Word docs or even RTF files, but it'll work nicely for plain text files.
- Uploading and downloading files should be supported; otherwise, we're limited to working on existing files and directories (or those we create), which would just be plain boring!
- When we select a directory from the tree, we should see the contents of that directory in a grid, showing us the item's name, size, type, and last modified date/time.
- Let's add one semi-advanced tool: the ability to print the contents of the current directory.
- Since a file manager is potentially a very dangerous application to run on a server, let's implement security using container-managed J2EE security. Let's be a bit more anal than we were with the DWiki project and protect all the relevant HTML/JSP resources, as well as the DWR-remotable classes and methods.
- The UI should look similar to Windows Explorer and also be reasonably extensible so that other capabilities can be added later.
- We'll want to avoid writing as much of the raw IO code as possible, so let's see what libraries are available to help us with this project.

This is a really nice project in terms of real-world usage because you can certainly see where installing it on a server would be very helpful. Clearly, you need to be careful because opening up the file system of a server is, to say the least, a dangerous thing to do. That being said, with some reasonable security precautions in the mix, it can prove to be a very handy remote management tool. The best part is we'll get to see some new facets to DWR in practical usage, and also get a glimpse of some nice UI components and libraries.

Speaking of which, let's start by taking a look at those UI components and those libraries. Even though these things aren't the focus of this book specifically, DWR is, they certainly do fit into the overall theme of Web 2.0 and Rich Internet Applications.

A BEVY OF FILE MANAGERS

I freely admit I'm a Windows guy. Yes, I actually **like** Windows! My *nix experience is fairly limited, although I've been learning a lot lately due to the environment I deal with on the job. I only point out these things because what I'm about to say is going to be strictly Windows oriented because I just don't have the same experience in other OSs to make similar statements there. In the immortal words of John Cleese in *A Fish Called Wanda*, I apologize unreservedly!

Windows Explorer gets the job done, but it's far from the best file manager out there. I've tried numerous Explorer replacements over the years, so I know a thing or two about what's available. What follows is a

short list of some of the better options I've found, and also my personal favorite. I encourage you to check them out, especially the first one on the list, because if my own experience is any indication, I believe you'll never be satisfied with Windows Explorer again!

- *Directory Opus—GP Software* (`www.gpsoft.com.au`): For me, no other file manager suffices! Just a few of the features that make Directory Opus the king for me: a dual-file display system, built-in file searching, directory compare feature, built-in file viewer in a dockable window, the ability to filter the display (either files or directories or both) by name, a built-in FTP client, the ability to copy just the names of all selected files to the clipboard (I can't tell you how handy that seemingly small feature is!), and the ability to show the sizes of all subdirectories of the current directory. Truly, this is one of the best pieces of software out there; I can't say enough about it. In fact, with Directory Opus and UltraEdit, I find I rarely need anything else to get most of my work done day in and day out.

- *xplorer² Lite—Nikos Bozinis* (`http://zabkat.com/x2lite.htm`): While Directory Opus is a commercial product, which is a turnoff for many, xplorer² Lite is free (at least there is a free edition, in addition to a paid version). xplorer² Lite actually has many of the features of Directory Opus and is thus certainly worth a look. It doesn't, in my opinion, have quite the same polish, but you can't argue with free!

- *Universal Explorer—Spadix Software* (`www.spadixbd.com/universal`): This is another commercial product that has a lot of nifty utility-type functions, things like disk copy, split and join of files, color list, ASCII list, and more. Universal Explorer has, to my eyes, a little bit of an outdated look, i.e., it doesn't have the updated Windows style that many apps do today. That shouldn't really dissuade you from it though; it's certainly got lots of features to make up for that (and you may well like that look more than the Fisher-Price look of Windows XP anyway).

- *FreeCommander—Marek Jasinski* (`www.freecommander.com`): Another free alternative, and a real dandy too. This actually has much of the power of Directory Opus as well, but for zero cents on the dollar. Definitely something I'd recommend a look at.

There are, of course, many other options, and fortunately I don't need to compile a larger list because someone has already done it: `www.simplehelp.net/2006/10/11/10-windows-explorer-alternatives-compared-and-reviewed/`.

This lists, in addition to the previously mentioned applications, a number of others, both free and commercial. I still recommend Directory Opus over all the others, because it just has so many features and feels the most polished, but many of the others are perfectly viable alternatives too. Whichever you like, the point is that replacing Windows Explorer, even though it's a serviceable file manager, will yield more productivity in the long run, cure cancer, and make Britney, Paris, and Lindsay all good citizens overnight. I promise.

Now stop laughing at the insanity of that last part and keep reading!

dhtmlx UI Components

I've had the good fortune of playing with a great many UI components over the past two years due to the nature of the applications I develop for a living. I've seen some real good ones and some real bad ones, and everything in between. Of all of them, the dhtmlx set of components stands out to me for a number of reasons.

The dhtmlx components (`www.dhtmlx.com`), which stands for DHTML eXtensions, are a product of a company called Scand LLC (`http://scbr.com`, based in Minsk, the capital of Belarus, formerly a part of the Soviet Union—see, I bet you never thought you'd be getting a geography and political science lesson here, but I aim to exceed your expectations!). Scand LLC provides offshore software development services, in addition to the set of components we'll be using in this project.

We'll be using four of the components this company offers, namely dhtmlxGrid, dhtmlxTree, dhtmlxToolbar, and dhtmlxMenubar. This is just a part of the overall offering though. Other components include dhtmlxTreeGrid (which, as you can guess, is a hybrid combination of the tree and grid components), dhtmlxTabbar (easily the best tabbed pane implementation I've ever seen), dhtmlxVault (an advanced file upload component), and dhtmlxCombo (a fancy drop-down combo component).

All of these components are offered under the open source GPL license, allowing you to use them free of charge in certain noncommercial situations. There are also paid versions that give you more features and support. Let me just point out that as good as these components are, they are exceedingly inexpensive: the dhtmlxTreeGrid is the most expensive at the time of this writing at $549 for an enterprise license (unlimited projects under one company) or $249 for a commercial license (a single project). Other components range in price from $49 to $149 for a commercial license (the enterprise licenses range from about $99 to $549). Let me tell you, for one project I did some months back, we chose a different component library (because we unfortunately hadn't discovered dhtmlx during our research phase) that cost in the $2,500 range for a more restrictive license, so in terms of cost, dhtmlx provides phenomenal value for a lot less coinage!

The dhtmlx components are well documented with ample working examples showing you how to do all sorts of things with them. I can't tell you how rare it is to find this level of quality around documentation and examples when it comes to client-side UI components. I've played with many that give you fits just figuring out how to get them on the screen, but you'll rarely if ever face such frustrations with the dhtmlx components.

It's important to realize that there are some important differences when you purchase a license. Aside from the simple fact that you get support, which certainly is important, all of the components gain functionality. For instance, the pro edition of the dhtmlxTree and dhtmlxGrid components give you more ways to work with large data sets, including smart parsing of tree data and smart rendering of the grid. For the grid, you gain the ability to freeze columns, use math formulas in cells, add footers, and perform CSV import/export. For the dhtmlxTabbar component, you gain the ability to have multiple lines of tabs and scrolling tabs if they overflow the area the tabs go in. This is just a small sample, mind you; feel free to explore at your leisure on the dhtmlx web site.

Let's take a look at each of these components, even the ones we won't be using in this project. They're definitely worth taking a look at.

In Figure 6-1, you can see the dhtmlxTree component.

Figure 6-1. *The dhtmlxTree component*

This is in fact two trees side by side inside a dhtmlxTabbar, but you get the idea. I should point out that all of these images are taken from the live examples on the dhtmlx web site, so if you go to the web site mentioned a few paragraphs back, you'll actually be able to play with this and the other examples. This one is nice because it shows the ability to drag and drop elements between the two trees, as well as the fully skinnable nature of all the components; in this case, just clicking a different tab brings up a whole different visual theme.

In Figure 6-2 is the dhtmlxGrid component. This is another fully "live" example, and you can do all sorts of whiz-bang things with it like select blocks of text and copy it to the clipboard, select multiple rows, edit cells, sort columns, and more.

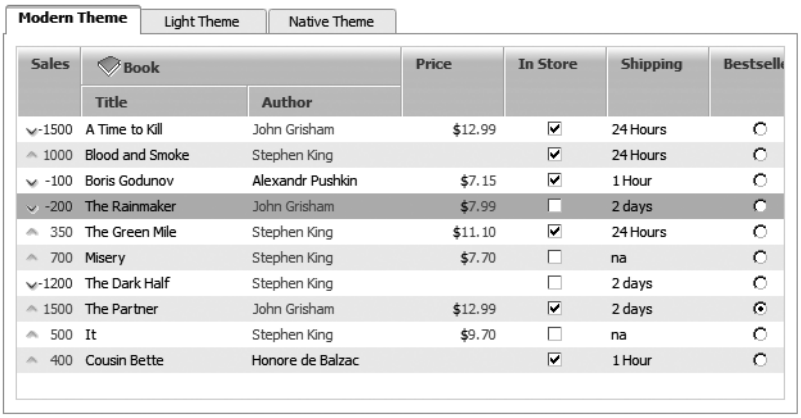

Modern Theme
Light Theme
Native Theme
Sales
Book
Price
In Store
Shipping
Bestsell
Title
Author
-1500 A Time to Kill John Grisham $12.99 24 Hours
1000 Blood and Smoke Stephen King 24 Hours
-100 Boris Godunov Alexandr Pushkin $7.15 1 Hour
-200 The Rainmaker John Grisham $7.99 2 days
350 The Green Mile Stephen King $11.10 24 Hours
700 Misery Stephen King $7.70 na
-1200 The Dark Half Stephen King 2 days
1500 The Partner John Grisham $12.99 2 days
500 It Stephen King $9.70 na
400 Cousin Bette Honore de Balzac 1 Hour

Figure 6-2. *The dhtmlxGrid component*

As with the dhtmlxTree example, you can switch the theme of the component easily, as demonstrated by the tabs once again.

The dhtmlxTreeGrid component is next, as shown in Figure 6-3.

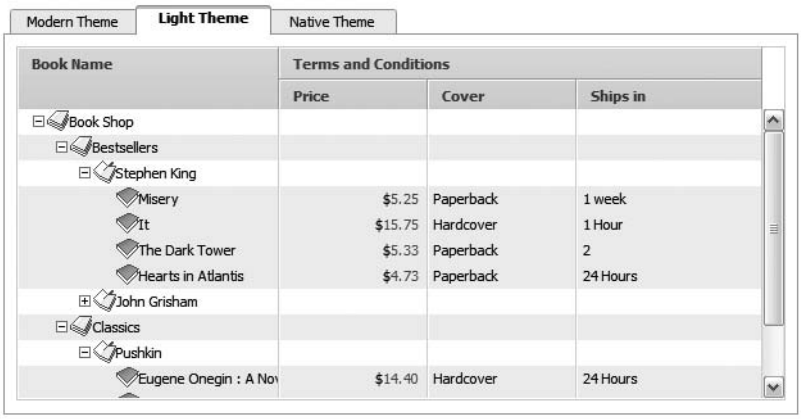


Figure 6-3. *The dhtmlxTreeGrid component*

This time, I've chosen a different theme for you to look at (unfortunately, in print, since there's no color, it won't look all **that** different, which is why a minute or two of your time spent on the web site would be worth it). Otherwise, this looks and works about how you'd expect a combination of a tree and a grid to work. It's not a typical user interface element, but one can imagine situations where it would come in handy (an exercise I leave to you and your imagination).

In Figure 6-4 is the dhtmlxTabbar component, which I have to again say is the single best implementation of tabs I've ever seen in a web UI component.

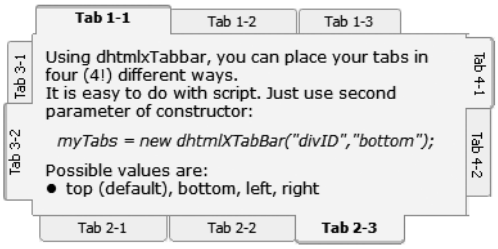


Figure 6-4. *The dhtmlxTabbar component*

You've already seen the dhtmlxTabbar in action actually, in the previous examples, but this one stands out because of the vertical tab capability. Few other implementations offer this capability at all, let alone as well done as this one is. I should mention that the dhtmlxTabbar is one of the instances where buying the professional/enterprise license is really worth a lot because it gives you some capabilities that I've never seen anywhere else, namely scrolling of the tabs when there are more than fit in the area you define for the tabs. Take a look at a Windows application where there are tons of tabs and the developer chose not to have multiple rows . . . in that case, you'll see little scrollbars to scroll the other tabs into view. The dhtmlxTabbar component offers that capability. Fantastic!

The next component to look at is the dhtmlxCombo, as shown in Figure 6-5, and it's probably the least interesting component frankly, but still very good and useful.

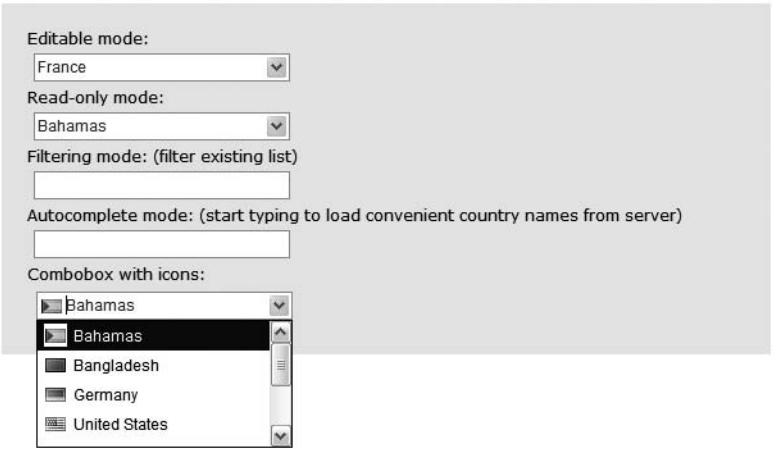


Figure 6-5. *The dhtmlxCombo component*

I've shown the bottom one expanded so that you can see the ability to put icons in the list. The first combo box actually allows editing as well as selecting of items, which is an ability that usually is not available to webapps. The others offer another common ability that until only recently wasn't typical in a webapp: filtering and searching of the list items. Type a G, for instance, and you'll be brought to the items beginning with G, in the case of the third combo. In the fourth, as you type, any item in the list matching what you've typed so far will automatically be completed for you.

In Figure 6-6 is the dhtmlxVault component, a very nifty extension to the typical HTTP-based file upload capability.

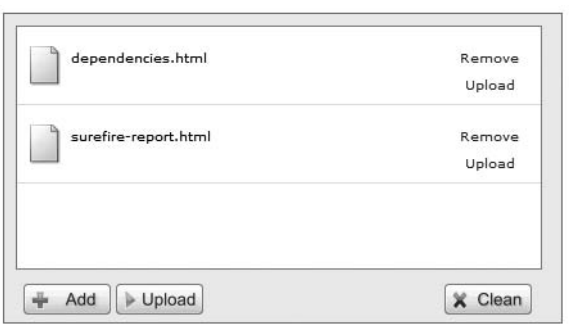


Figure 6-6. *The dhtmlxVault component*

Users of Firefox will feel right at home as this looks an awful lot like the file download manager seen there, at least as of Firefox 2.*x* (I hear in version 3.0 it'll look a bit different). As you click the Add button, the files you select are added to the vault. Clicking Upload uploads each file, in sequence, and each time there is a nice little status bar that grows as the file uploads.

In Figure 6-7 is the dhtmlxToolbar component.

Figure 6-7. *The dhtmlxToolbar component*

As you can see, the dhtmlxTabbar is very customizable, providing not only simple buttons, but also `<select>` elements and even a label on the right. Incidentally, all of these components can be built programmatically, or also by feeding a string of XML to a new instance, so you can do completely declarative creation of these components. Very handy indeed!

Finally, in Figure 6-8, you can see the last component offered, which is the dhtmlxMenubar component.

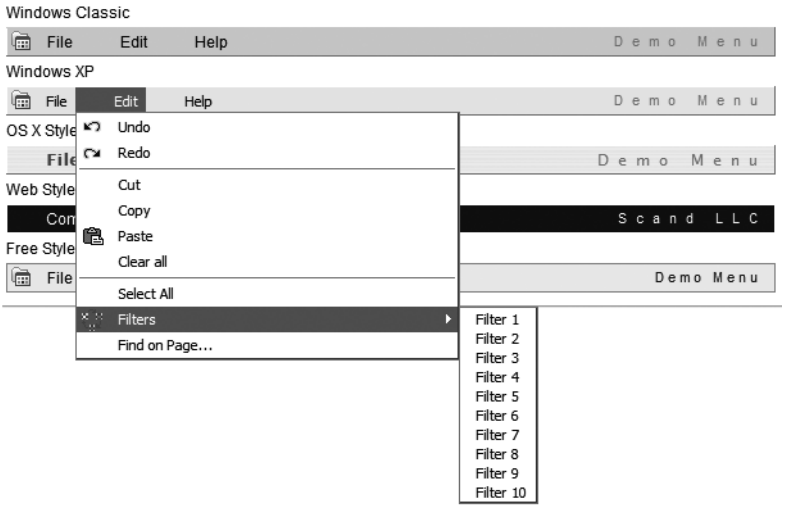


Figure 6-8. *The dhtmlxMenubar component*

Here, you can see the multiple skins available, as well as how the menus can contain icons, multilevel submenus, and a label just like the dhtmlxTabbar.

There is also a dhtmlxFoldersPane that will probably be available as you read this (it was still in development at the time of this writing). It ironically would have made the application in this chapter that much better because it provides the functionality of various views such as thumbnails as you see in Windows Explorer.

The actual usage of these controls we'll see as we go through the code of the project, but in short it's frankly about what you'd expect. Want to create a toolbar? Just do this:

```
toolbar = new dhtmlXToolbarObject(
  dwr.util.byId("divToolbar"), "100%", "20px", "");
```

We just tell it what element on the page to create it in, the `<div>` named `divToolbar` here, and also specify its width and height, and label (none here), and that's it. If we then want to add a button to it, things couldn't be simpler:

```
var tbRefresh = new dhtmlXImageButtonObject(
  "img/icoRefresh.gif", "38px", "32px", null, "tbRefresh",
  "Refresh current directory", null, disableImage="img/icoRefresh.gif");
toolbar.addItem(tbRefresh);
```

Lastly, we need to actually show the toolbar, and it takes only a single line of code to accomplish that:

```
toolbar.showBar();
```

That's the programmatic approach to things. You can also do it all with XML. Given the following XML:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<toolbar name="Demo toolbar" width="100%" disableType="image"
  absolutePosition="auto" toolbarAlign="left">
        <ImageButton src="img/icoRefresh.gif" height="38" width="32" id="tbRefresh"
        tooltip="Refresh current directory" />
</toolbar>
```

the following code can be used:

```
Toolbar = new dhtmlXToolbarObject(dwr.util.byId("divToolbar"), "100%", "20px","");
toolbar.loadXML("toolbar.xml")
toolbar.showBar();
```

It comes down pretty much to preference in deciding which approach you want to use. The XML approach necessitates an extra hit on the server (loading the XML file), but the programmatic approach involves, well, more **programming**. If you're going to be dynamically altering the toolbar as the application runs, you might be inclined to go with the programmatic approach from the beginning, but that's entirely up to you.

I hope you agree that the dhtmlx UI components are indeed very nice, to say the least. The price is right, the documentation and examples are top-notch, they look and function as good or better than any alternative I've seen, and by the way, the folks at Scand LLC are very nice to deal with, very helpful and responsive to queries. I'm not a paid shill for them, but I'm the next best thing: a happy customer who thinks the company offers a great product at a good price, with good service to go along with it all. As you'll see in this project, these components help make for an outstanding web UI with a minimum of effort, exactly as you'd want such a product to do.

RICH WEB-BASED USER INTERFACES: NOT JUST A DREAM ANY MORE!

It used to be, not very long ago even, that if you wanted to have a decent user interface, the Web wasn't what you wanted to look at. HTML form controls, to this day, are pretty limited, and even as people started to work around those limitations a bit, especially with the advent of CSS and JavaScript, the limitations were still very much there.

Over the last year or two though, we've started to see something new: the advent of powerful UI components for web developers. This has come about along with the whole Ajax and Web 2.0 fad because, I think, people have finally become comfortable with the idea of JavaScript and client-side development.

This new breed of UI components are generally not just spiffed-up HTML form controls, but are instead complete UI widgets built from smaller pieces, glued together with liberal amounts of JavaScript. One of the biggest names in this area is the Dojo toolkit, and we'll actually get a glimpse of that in the next chapter (if you'd like to jump ahead and have a look now, cruise on over to `http://dojotoolkit.org`, but hurry back!). Dojo goes so far as to provide a standardized framework from which widgets are built, so they all look, from a coding standpoint, somewhat similar. This is virtually unheard of in the world of web development.

The bottom line, regardless of which library or toolkit you use, is that truly rich, fat-client-like web UIs are now not just flights of fancy, but are being produced regularly now by most development staffs. You no longer have to choose between a web-based application and a good UI; both are now firmly within reach. You no longer have to tell your clients, "Sure, we can do a web application, and get all the benefits they provide, but you have to sacrifice a good-looking, rich user experience." Nope, not any more; you can finally have your cake and eat it too, as I hope this project shows. It may not be Mac OS–like in its UI coolness, but I don't think the fact that it's a webapp hurts it any either in terms of the UI richness, and that's the whole point to this tangent.

Jakarta Commons IO

Writing raw file IO can be a drag, be it in Java or any other language. Oh sure, some operations are simpler than others, but some are just a royal pain in the rump. Fortunately, with Commons IO helping, you'll find your rump is far less angry at you at the end of the day.

Commons IO (`http://commons.apache.org/io`) provides a relatively small number of packages that contain a lot of power. Table 6-1 lists those packages and briefly describes each.

Table 6-1. *A Breakdown of the Packages Provided by Commons IO*

Package	Description
`org.apache.commons.io`	This package defines utility classes for working with streams, readers, writers, and files.
`org.apache.commons.io.filefilter`	This package defines an interface (`IOFileFilter`) that combines both `FileFilter` and `FilenameFilter`.
`org.apache.commons.io.input`	This package provides implementations of input classes, such as `InputStream` and `Reader`.
`org.apache.commons.io.output`	This package provides implementations of output classes, such as `OutputStream` and `Writer`.

In this project, we'll be using classes from only the first, `org.apache.commons.io`. The specific classes we'll be using are

- `DirectoryWalker`: This is an abstract class that we'll need to extend. Its job is to walk down a particular directory hierarchy, providing subclasses with callback hooks to react to various events that occur during that walking. We'll be using it to get a list of directories contained within a starting directory.
- `FileUtils`: This will be used to provide the capabilities of creating, editing, and saving files, as well as copying files.

Although that doesn't look like much, it makes the code we're required to write considerably less verbose, and at least in theory makes it less error prone too. There were probably some opportunities to use other functions, but frankly, most of the other functions are handled quite simply by basic Java `File` class functions, so there wasn't quite as much gain as with these situations.

Jakarta Commons FileUpload

If you've ever looked at the spec[1] for supporting HTTP-based file uploads, you'll know that it's not a trivial thing to implement properly and robustly. If, in years gone by, you were crazy enough (like I was at one point) to try and implement it yourself, you'll have an appreciation for a library that gives you a good, solid implementation that is easy to use.

Well, Commons FileUpload is exactly such an implementation.

FileUpload makes it almost too easy to handle uploads. Probably the simplest usage is the following code, which I think is pretty self-explanatory and obvious:

```
FileItemFactory factory = new DiskFileItemFactory();
ServletFileUpload upload = new ServletFileUpload(factory);
List items = upload.parseRequest(request);
```

This results in a `List` of `FileItem` objects. Each `FileItem` may actually represent an uploaded file or another nonfile form field submitted as part of the form. So, you can do something like the following:

```
while (items.iterator().hasNext()) {
  FileItem item = (FileItem)it.next();
  if (item.isFormField()) {
    System.out.println(item.getFieldName() + " = " + item.getString());
  } else {
    // Save file to disk
  }
}
```

That will output to `stdout` each nonfile form field and its value. When a file field is detected, the actual implementation of saving the file to disk can be anything you want, but one simplistic approach is this:

```
File uploadedFile = new File("path_and_filename_here");
item.write(uploadedFile);
```

That's really all it takes! As I said, this is probably the most simplistic usage possible; there are in fact many more ways to do things with FileUpload. For our purposes here, however, we won't get a whole lot more complicated than this.

---

1. The HTTP form-based file upload spec, more properly called RFC 1867, can be seen here: `www.ietf.org/rfc/rfc1867.txt`. When you've completed wiping away the cold sweat reading that is likely to give you, come on back and forget you ever looked at it!

WHEN LAZY IS GOOD: USE THOSE LIBRARIES!

I know I've said this a few times throughout this book, and this likely won't be the last time, but if you aren't regularly checking out the goings-on at Apache's Jakarta Commons (`http://commons.apache.org`), you're working a lot harder than you probably need to! Apache has produced some great things over the years, from the Apache server itself to Tomcat to Struts and others, but Commons is perhaps one of the most consistently useful projects under the Apache umbrella.

In case you aren't familiar with Commons, it began as a place where parts of other Apache projects that developers have recognized as being generically useful are put. For instance, one component called Commons Digester was originally a part of the Struts framework that was developed to parse the `struts-config.xml` file. Digester allows you to map XML files to Java objects and have a graph of objects created as the XML is parsed, which is exactly what's needed when parsing the Struts configuration file. As a result, the Struts developers realized this could be a very useful tool outside of Struts, so it was moved to Commons.

These days, not all of Commons emerges from other Apache projects; a few Commons components are donated from outside sources, or began as pet projects of existing Apache developers. However a component comes to Commons, though, the goal is the same: reusable Java components that save developers from having to write the same code over and over. And Commons has proved a resounding success in fulfilling those goals.

There are a couple of mailing lists for Jakarta Commons, most of the subcomponents have their own, and Commons as a whole has at least one. These are good sources of information about what is coming down the pike. Also note that Commons uses the concept of a sandbox, where new components that have not officially been brought into Commons can be found. Every now and again I take some time to see what's brewing there, which I certainly recommend to any Java developer.

Also note that as with all Apache projects, contributions from fellow developers are welcome. I myself have had a few contributions accepted into Commons, so don't hesitate if you have something to contribute!

Dissecting Fileman

We now begin our exploration of the Fileman application, beginning with a look at its directory structure (see Figure 6-9). As usual, it's an unremarkable Java-based webapp by and large, nothing too surprising.

In the root directory, you'll find three HTML files and three JSP files. These make up the application itself, and we'll look at each of them in turn as we dissect the application.

I have chosen not to expand the `img` directory, as it contains a bunch of images used by the various dhtmlx components, too numerous to discuss here.

In the `css` directory, we have a number of style sheets, one each for the four dhtmlx components we're using, and the main `styles.css` used by the rest of the application.

In the `js` directory, we have all the JavaScript files necessary for all the dhtmlx components, as well as the main `Fileman.js` file that constitutes the client-side code of the application. Note that looking at the code for the dhtmlx components is well out of the scope of this book, so little more will be said about them. It's enough to know that they are necessary for the components to function, and we'll leave it at that.

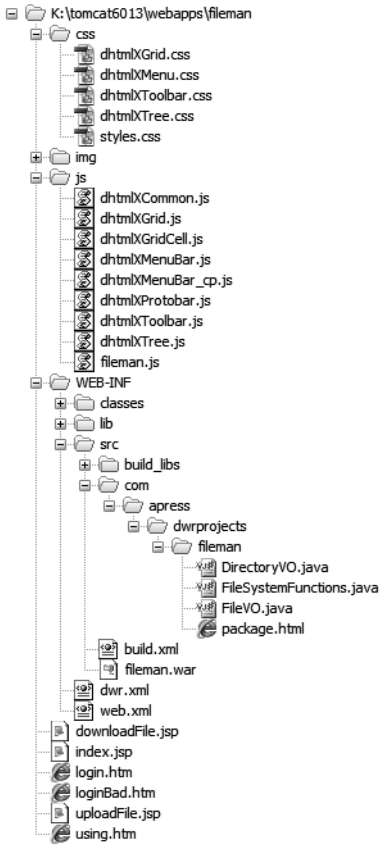


Figure 6-9. *The directory structure of the application*

The `WEB-INF` directory contains the typical contents, namely `web.xml` and `dwr.xml`. The `classes` and `lib` directories are also not expanded here to save some space. In the `lib` directory, you'll find what we've seen before: `dwr.jar` and `commons-logging.jar`. You'll also find `commons-io.jar` and `commons-fileupload.jar`, the two new Commons libraries we need for this application.

We also have our `src` directory under `WEB-INF`, and this has the same basic structure as all the other applications in this book, so no point going over it in any detail here.

With the directory structure understood, let's now take our first look at Fileman itself. In Figure 6-10, you can see what it looks like when first started up, before any directory has been selected.

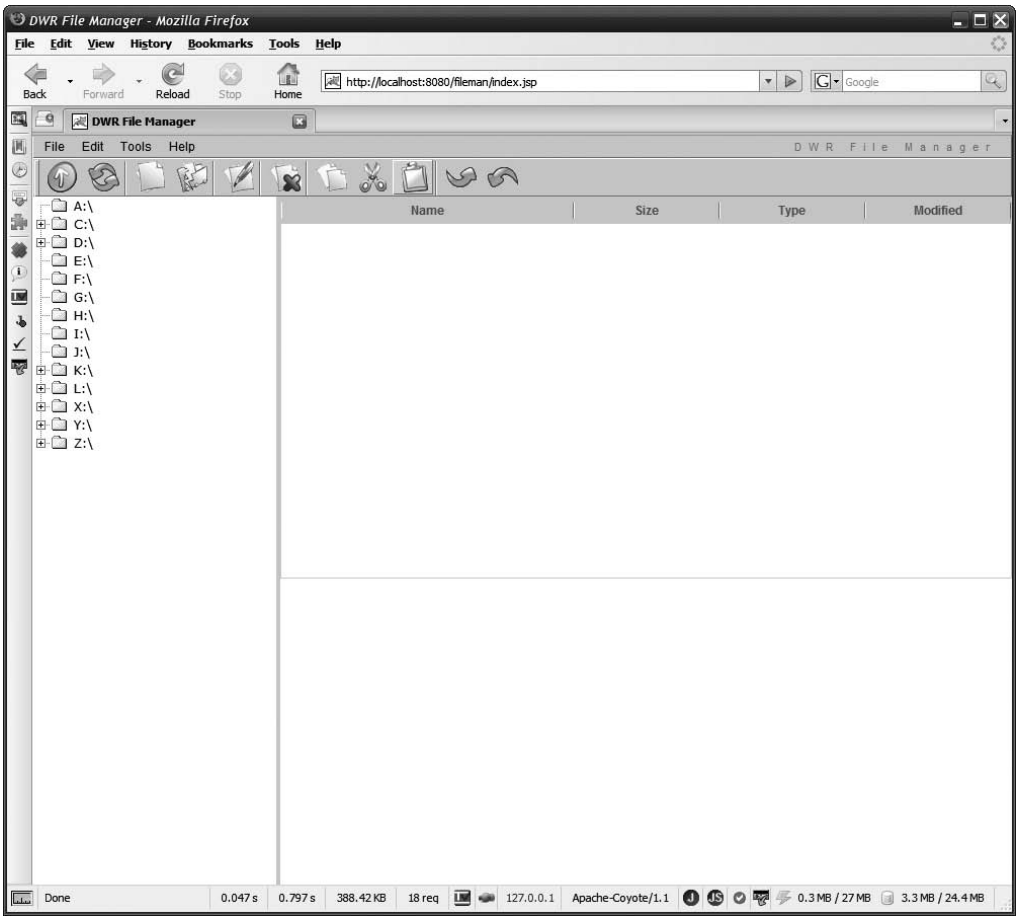


Figure 6-10. *A first glimpse of Fileman*

You can see the basic structure of the interface here: menu bar on top, with a toolbar below that, and the directory tree on the left and the file grid on the right. Once again, this is very much similar to Windows Explorer, and indeed most other Windows file managers (and probably most file managers on any other operating system too). From left to right, the toolbar buttons are

- Up one level
- Refresh current directory
- Create new file
- Create new directory
- Edit file
- Delete file
- Copy file or directory

- Cut file or directory
- Paste file or directory
- Upload file
- Download file

The menu contains all these options, plus a few more, namely the Print Directory Contents on the Tools menu, and the Using Fileman and About items on the Help menu.

We'll see more snapshots of the application in action as we progress, but for now that should be enough to have a clue about what we're building. So, we're all set to jump into the code now, and a reasonable place to start is the configuration files involved, so off we go!

In all the following source listings, I've removed the `GPL/copyright` header to save space and cut down on redundancy. The license terms of the dhtmlx components require that the application they are used in be licensed under the terms of the GPL if it's noncommercial, which is the case here. Even though the kind folks at Scand LLC actually gave me permission to not have to do so for the purposes of this book, I felt it was better to release this code under the GPL anyway so that there were no potential legal issues for anyone, reader and author/publisher alike. This also means that you'll very likely find this in the sample chapter distributed on the web for promotional purposes.

Configuration Files

As we saw in the DWiki project, there are three configuration files involved when you're dealing with J2EE security: the usual `web.xml` and `dwr.xml` files, plus the configuration file specific to the container the application is running in, `tomcat-users.xml` in the case of Tomcat.

Container-Managed Security Configuration

If you've already read the previous chapter and have looked at the DWiki application, this will be nothing new at all to you. If you haven't read that chapter yet, I suggest going back and at least reading the section entitled "Container-Managed Security Configuration." In Listing 6-1, we see the relevant configuration data that's required.

Listing 6-1. *The* `tomcat-users.xml` *File*

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="fileman_user"/>
  <user username="file" password="man" roles="fileman_user"/>
</tomcat-users>
```

This is a simple configuration consisting of a single group and a single user, named "file", with a horribly insecure password of "man". Should you decide to put this on your production server as-is, I cannot be held responsible for the fate of your data! That being said, for the sake of a nice, simple example, this should be fine.

web.xml

In Listing 6-2, we can see the `web.xml` file for Fileman. It is very much along the lines of all the other configuration files throughout the book, and the security setup is very much similar to that of DWiki again, although a bit more verbose.

Listing 6-2. *The* `web.xml` *Application Configuration File*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app id="fileman" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>FileMan</display-name>

  <!-- DWR servlet. -->
  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
    </init-param>
  </servlet>

  <!-- Servlet mapping. -->
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

  <!-- Security setup. -->
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.htm</form-login-page>
      <form-error-page>/loginBad.htm</form-error-page>
    </form-login-config>
  </login-config>
  <security-constraint>
    <web-resource-collection>
```

```
    <url-pattern>/index.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>fileman_user</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/downloadFile.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>fileman_user</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/uploadFile.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>fileman_user</role-name>
  </auth-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <url-pattern>/using.htm</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>fileman_user</role-name>
  </auth-constraint>
</security-constraint>
<security-role>
  <role-name>fileman_user</role-name>
</security-role>

<!-- Session timeout config. -->
<session-config>
  <session-timeout>30</session-timeout>
</session-config>
```

```
  <!-- Welcome file config. -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

With regard to the security setup, the way I chose to do it is to be as explicit as possible, so I actually protected each and every HTML and JSP file, save those used during login. Things could have been organized differently, perhaps in a subdirectory where all the protected resources go, which would have reduced all of that to a single constraint, but this way there's no doubt about a mapping being incorrect. Also note that unlike DWiki, I've constrained both HTTP `GET` and `POST` methods. Especially for the file upload JSP, this is important because if just `GET` were constrained, then a nonauthorized user would still be able to upload a file! That clearly would not be a good thing, and that goes for downloading files more so, which is why that JSP is similarly protected (in fact, they **all** are constrained on both methods, which is probably a little overly anal; but again, this is a potentially very dangerous application, so we want to be as vigilant as possible).

All the other aspects of this `web.xml` file are things you've seen before in other projects and will continue to see in most of them to follow, so I won't spend any time on the rest here. Please do review previous project chapters if you have any doubts about anything here.

dwr.xml

By now I suspect that `dwr.xml` is second nature to you, and there aren't any real surprises in the version for Fileman, as shown in Listing 6-3.

Listing 6-3. *The DWR Configuration File for Fileman,* `dwr.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>

    <!-- Need to be able to receive FileVO and DirectoryVO objects from -->
    <!-- the server into JavaScript objects. -->
    <convert converter="bean"
      match="com.apress.dwrprojects.fileman.DirectoryVO" />
    <convert converter="bean" match="com.apress.dwrprojects.fileman.FileVO" />

    <!-- Converter for dealing with exceptions. -->
    <convert match="java.lang.Exception" converter="exception" />

    <!-- Only a single class is necessary for this application. -->
    <create creator="new" javascript="FileSystemFunctions">
```

```
      <param name="class"
        value="com.apress.dwrprojects.fileman.FileSystemFunctions" />
      <!-- By default, because this is a potentially dangerous application, -->
      <!-- we want to exclude all methods of the FileSystemFunctions class. -->
      <!-- Therefore, we need to "expose" the methods that are needed -->
      <!-- by the client.  In addition, each will be secured via J2EE -->
      <!-- security for added safety. -->
      <include method="listFiles" />
      <auth method="listFiles" role="fileman_user" />
      <include method="listRoots" />
      <auth method="listRoots" role="fileman_user" />
      <include method="listDirectories" />
      <auth method="listDirectories" role="fileman_user" />
      <include method="renameFile" />
      <auth method="renameFile" role="fileman_user" />
      <include method="copyMoveFile" />
      <auth method="copyMoveFile" role="fileman_user" />
      <include method="deleteFile" />
      <auth method="deleteFile" role="fileman_user" />
      <include method="createFile" />
      <auth method="createFile" role="fileman_user" />
      <include method="createDirectory" />
      <auth method="createDirectory" role="fileman_user" />
      <include method="editFile" />
      <auth method="editFile" role="fileman_user" />
      <include method="saveFile" />
      <auth method="saveFile" role="fileman_user" />
    </create>

  </allow>
</dwr>
```

`DirectoryVO`s and `FileVO`s are returned by the server for a couple of operations, thus we need converters defined for them so DWR knows it can marshal them to JavaScript objects, and how. As is true in most cases, the basic bean converter does the job nicely for us.

Since there is only a single class that is to be remoted, `FileSystemFunctions`, only a single `<creator>` entry is required. However, because we're trying to be very security conscious here, this single entry also has a whole bunch of `<include>` and `<auth>` entries.

In order to make this application as safe as possible, rather than allowing DWR to remote any and all methods in the `FileSystemFunctions` class, as is the default way DWR works, we instead want to only allow remoting of those methods we specifically list. That's the purpose of the `<include>` methods. Once you have a single one of these elements under a `<creator>` element, you're effectively telling DWR, "Hey, no methods are remotable *except* those I specify here." Now, in the case of `FileSystemFunctions`, we're opening up nearly all of them anyway, but this is still a good practice. In addition to specifying which methods to remote, they are also locked down with J2EE security, so only authenticated users in the `fileman_user` group will be allowed to execute them.

With those two restrictions in play, this application should be fairly secure. It's probably not going to keep the NSA[2] (or KGB, or MI-5, or whatever the equivalent organization in your country is) out of your business, but your little brother Billy shouldn't be able to get in.

On second thought, little Billy is probably the bigger threat in this day and age . . . these kids today and their mad haxx0r[3] skills are beyond what I did in my foolish youth!

The Client-Side Code

We're now ready to start looking at the actual code behind Fileman. As with other projects, we'll begin by looking at the client side of things. First up is the main style sheet for the application, `styles.css`.

styles.css

The `styles.css` file is the main style sheet for the Fileman application. Recall that there are a number of other style sheets in the `css` directory for the dhtmlx components, but we won't be looking at them here as they are by and large out of the scope of this book.

In Listing 6-4, you can see the full source for the `styles.css` file.

Listing 6-4. *The Style Sheet for Fileman, As Housed in* `styles.css`

```
/* Style applied to the body of the main index.jsp page. */
.cssBody {
  padding          : 0px;
  margin           : 0px;
  overflow         : hidden;
}


/* Style applied to the div housing the menubar. */
.cssMenu {
  width            : 100%;
  height           : 22px;
  background-color : #eaeaea;
}
```

2. In the United States, the NSA is the National Security Agency, one of the seemingly endless intelligence agencies we have here. The KGB, which is the Russian-language abbreviation for Committee for State Security, is (roughly) the Soviet-era equivalent (the KGB has now been superseded by the FSB, Federal Security Service, and don't ask me where the B came from!). MI-5, the Security Service, is Britain's equivalent (there is an MI-6 as well, which may or may not be a closer equivalent . . . the difference wasn't quite clear to me reading Wikipedia, which I suppose is how you'd want your spooks: a little mysterious and difficult to comprehend).

3. Haxx0r is Leetspeak (1337) for hacker. Leetspeak is a way of speaking, or more precisely, writing, words, usually associated with the computer "underground" (software pirates, hackers, crackers, hardcore gamers, and so on).

```
/* Style applied to the div housing the toolbar. */
.cssToolbar {
  width            : 100%;
  height           : 38px;
}


/* Style applied to the vertical divider between the directory tree and */
/* the file grid. */
.cssDivider {
  background-color : #d4d0c8;
}


/* Style applied to the div housing the directory tree. */
.cssDirectories {
  width            : 100%;
  height           : 100%;
  overflow         : hidden;
}


/* Style applied to the div housing the file grid. */
.cssFiles {
  width            : 100%;
  height           : 100%;
  overflow         : hidden;
}


/* Style applied to the file upload and file editor divs. */
.cssFullScreenDiv {
  width            : 100%;
  height           : 100%;
  background-color : #ffffff;
  display          : none;
  z-index          : 1000;
  position         : absolute;
  left             : 0px;
  top              : 0px;
}
```

The `cssBody` class ensures that the full browser content area is available for the application by setting both margin and padding to zero, and removing the scrollbars.

The `cssMenu` class is applied to the `<div>` housing the dhtmlxMenubar component, so that we know precisely how much space it'll take up. The `cssToolbar` class is likewise applied to the `<div>` housing the dhtmlxToolbar component and serves the same sizing purpose.

The `cssDivider` class is applied to the table column that serves as the divider between the directory tree and the file grid.

The `cssDirectories` class is applied to the table column where the directory tree is, and perhaps its most important job is to ensure that there are no scrollbars there (they are provided by the dhtmlxTree component itself, so we want to ensure we don't get two scrollbars, one on the containing element and one on the component itself). The `cssFiles` class is likewise applied to the column where the file grid is and serves the same purpose.

The `cssFullScreenDiv` class is last, and it is used for the file upload form, as well as the file edit form. These two displays are simply `<div>`s sized to fill the entire browser content area, with a `z-index` that puts it above everything else. This `<div>` is positioned absolutely at the upper-left corner of the browser content area. The result is that when either of these `<div>`s is shown, with this style class applied, they cover everything else and look like the only thing in the browser at that time.

login.htm

The first of the six HTML/JSP pages we're going to look at is `login.htm`. You can see what this page actually looks like in Figure 6-11.

DWR File Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/fileman/
Google
DWR File Manager
Please log in
Username:
Password:
Login
0.125 s 0.172 s 797 bytes 2 req 127.0.0.1 Apache-Coyote/1.1 0.3 MB / 27 MB 3.3 MB / 24.4 MB

Figure 6-11. *The amazingly dull login page of Fileman*

It's not much to look at, but it gets the job done. When the user accesses the application, and he or she is not yet authenticated, that user is redirected to this page. It's a typical J2EE login page, as you can see in Listing 6-5.

Listing 6-5. *The* `login.htm` *File*

```
<html>
  <head>
    <title>DWR File Manager</title>
  </head>
  <body>
```

```
    <h1>Please log in</h1>
    <hr>
    <br>
    <form name="j_security_check" method="post" action="j_security_check">
      <table border="0" cellpadding="2" cellspacing="2">
        <tr>
          <td width="1">Username: </td>
          <td><input type="text" name="j_username" value="" size="11"
            maxlength="10"></td>
        </tr>
        <tr>
          <td>Password: </td>
          <td><input type="password" name="j_password" value="" size="11"
            maxlength="10"><br>
        </tr>
        <tr>
          <td colspan="2" align="right">
            <input type="submit" value="Login">
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

It's nothing more than a simple form that submits to the standard `j_security_check` servlet. The `j_username` and `j_password` fields are all that are required. There wasn't even any point in making this a JSP, since it actually is nothing but straight markup, better to avoid the added processing required to execute a compiled JSP.

loginBad.htm

After the form on `login.htm`, the container attempts to validate the credentials entered by the user. If they are found to be valid, the page originally requested is served (which would be `index.jsp`, the default page defined in `web.xml`). If the credentials are found to not be valid, however, the user is presented with the page saying he or she could not be logged in. Just for the sake of being complete, the source code for this page is shown in Listing 6-6.

Listing 6-6. *The* `loginBad.htm` *File*

```
<html>
  <head>
    <title>DWR File Manager</title>
  </head>
  <body>
    Login incorrect.  Please <a href="index.jsp">try again</a>.
  </body>
</html>
```

There is no screenshot for this page because it's nothing but a single line of text with a link the user clicks to retry the login. Note that the link goes to `index.jsp`, which means the container will again intercept the request and redirect to `login.htm`, so our whole login mechanism is consistent before and after a failed login attempt.

index.jsp

Now we come to the main markup page for Fileman, namely `index.jsp`, which again is where the user will wind up after a successful login attempt. We saw what Fileman looks like initially earlier, so now let's take another look at it, this time showing the contents of my `Windows` directory, in Figure 6-12.

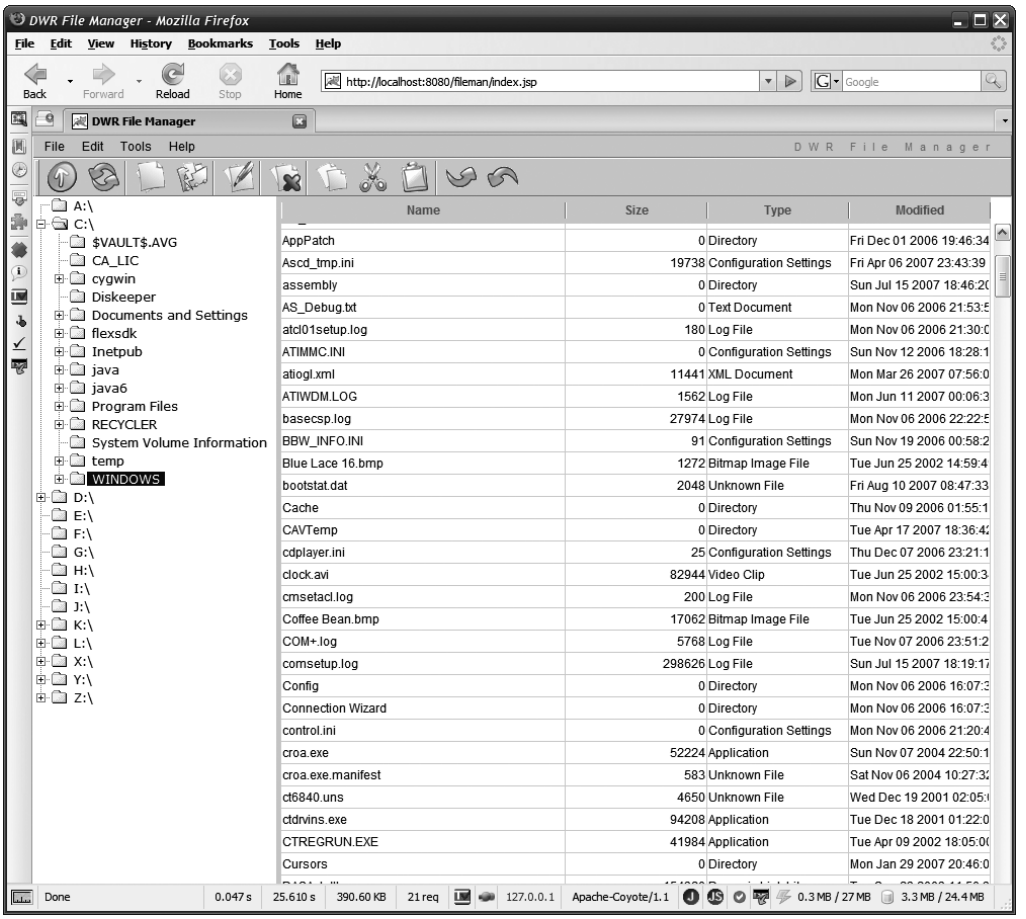

DWR File Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/fileman/index.jsp
Google
DWR File Manager
File Edit Tools Help
DWR File Manager
A:\
C:\
$VAULT$.AVG
CA_LIC
cygwin
Diskeeper
Documents and Settings
flexsdk
Inetpub
java
java6
Program Files
RECYCLER
System Volume Information
temp
WINDOWS
D:\
E:\
F:\
G:\
H:\
I:\
J:\
K:\
L:\
X:\
Y:\
Z:\
Name Size Type Modified
AppPatch 0 Directory Fri Dec 01 2006 19:46:34
Ascd_tmp.ini 19738 Configuration Settings Fri Apr 06 2007 23:43:39
assembly 0 Directory Sun Jul 15 2007 18:46:20
AS_Debug.txt 0 Text Document Mon Nov 06 2006 21:53:5
atcl01setup.log 180 Log File Mon Nov 06 2006 21:30:0
ATIMMC.INI 0 Configuration Settings Sun Nov 12 2006 18:28:1
atiogl.xml 11441 XML Document Mon Mar 26 2007 07:56:0
ATIWDM.LOG 1562 Log File Mon Jun 11 2007 00:06:3
basecsp.log 27974 Log File Mon Nov 06 2006 22:22:5
BBW_INFO.INI 91 Configuration Settings Sun Nov 19 2006 00:58:2
Blue Lace 16.bmp 1272 Bitmap Image File Tue Jun 25 2002 14:59:4
bootstat.dat 2048 Unknown File Fri Aug 10 2007 08:47:33
Cache 0 Directory Thu Nov 09 2006 01:55:1
CAVTemp 0 Directory Tue Apr 17 2007 18:36:42
cdplayer.ini 25 Configuration Settings Thu Dec 07 2006 23:21:1
clock.avi 82944 Video Clip Tue Jun 25 2002 15:00:3
cmsetacl.log 200 Log File Mon Nov 06 2006 23:54:3
Coffee Bean.bmp 17062 Bitmap Image File Tue Jun 25 2002 15:00:4
COM+.log 5768 Log File Tue Nov 07 2006 23:51:2
comsetup.log 298626 Log File Sun Jul 15 2007 18:19:17
Config 0 Directory Mon Nov 06 2006 16:07:3
Connection Wizard 0 Directory Mon Nov 06 2006 16:07:3
control.ini 0 Configuration Settings Mon Nov 06 2006 21:20:4
croa.exe 52224 Application Sun Nov 07 2004 22:50:1
croa.exe.manifest 583 Unknown File Sat Nov 06 2004 10:27:32
ct6840.uns 4650 Unknown File Wed Dec 19 2001 02:05:0
ctdrvins.exe 94208 Application Tue Dec 18 2001 01:22:0
CTREGRUN.EXE 41984 Application Tue Apr 09 2002 18:05:0
Cursors 0 Directory Mon Jan 29 2007 20:46:0
Done 0.047 s 25.610 s 390.60 KB 21 req 127.0.0.1 Apache-Coyote/1.1 0.3 MB / 27 MB 3.3 MB / 24.4 MB

Figure 6-12. *Fileman as it appears when viewing my* `Windows` *directory*

So, now that we have a good idea what Fileman looks like both initially and when a directory is selected for viewing, let's tear into the markup that makes this all happen. Listing 6-7 shows the contents of `index.jsp` in its entirety.

Listing 6-7. *The Main Client-Side Markup of Fileman,* index.jsp

```
<%@ page import="java.io.File" %>

<%
  // Get the system-specific path separator, we'll need it later.
  String pathSeparator = File.separator;
  if (pathSeparator.equals("\\")) {
    pathSeparator = "\\\\";
  }
%>

<html>
  <head>
    <title>DWR File Manager</title>

    <!-- Import all Javascript needed. -->
    <script src="dwr/interface/FileSystemFunctions.js"></script>
    <script src="dwr/engine.js"></script>
    <script src="dwr/util.js"></script>
    <script src="js/Fileman.js"></script>
    <script src="js/dhtmlXCommon.js"></script>
    <script src="js/dhtmlXGrid.js"></script>
    <script src="js/dhtmlXGridCell.js"></script>
    <script src="js/dhtmlXTree.js"></script>
    <script src="js/dhtmlXProtobar.js"></script>
    <script src="js/dhtmlXToolbar.js"></script>
    <script src="js/dhtmlXMenuBar.js"></script>
    <script src="js/dhtmlXMenuBar_cp.js"></script>

    <!-- Import all stylesheets needed. -->
    <link rel="stylesheet" type="text/css" href="css/dhtmlXGrid.css">
    <link rel="stylesheet" type="text/css" href="css/dhtmlXTree.css">
    <link rel="stylesheet" type="text/css" href="css/dhtmlXToolbar.css">
    <link rel="stylesheet" type="text/css" href="css/dhtmlXMenu.css">
    <link rel="stylesheet" type="text/css" href="css/styles.css">

  </head>

  <body class="cssBody" onLoad="fileman.init('<%=pathSeparator%>');"
    onResize="fileman.onResize();">

    <!-- File editor. -->
    <div id="divFileEditor" class="cssFullScreenDiv" style="display:none;">
      <table border="0" cellpadding="0" cellspacing="0" width="100%"
        height="100%">
        <tr><td align="center" valign="middle">
          <textarea id="taFileEditor" rows="20" cols="100"></textarea>
```

```
      <br>
      <input type="button" value="Save File" onClick="fileman.saveFile();">
          
      <input type="button" value="Cancel"
        onClick="dwr.util.byId('divFileEditor').style.display='none';">
    </td></tr>
  </table>
</div>

<!-- Upload file. -->
<div id="divFileUpload" class="cssFullScreenDiv" style="display:none;">
  <table border="0" cellpadding="0" cellspacing="0" width="100%"
    height="100%">
    <tr><td align="center" valign="middle">
      <h1>Upload a file</h1>
      <br>
      <form action="uploadFile.jsp" method="post"
        enctype="multipart/form-data" target="_new">
        <input type="hidden" name="uploadDirectory">
        File to upload: <input type="file" name="theFile">
        <br><br>
        <input type="submit" value="Upload file">
            
        <input type="button" value="Cancel"
          onClick="dwr.util.byId('divFileUpload').style.display='none';">
      </form>
    </td></tr>
  </table>
</div>

<table width="100%" height="100%" border="0" cellpadding="0"
  cellspacing="0">

  <!-- Menu and toolbar. -->
  <tr>
    <td colspan="3" height="60" align="left" valign="top">
      <div class="cssMenu" id="divMenu"></div>
      <div class="cssToolbar" id="divToolbar"></div>
    </td>
  </tr>

  <!-- Directory tree and file grid. -->
  <tr>
    <td width="25%">
      <div class="cssDirectories" id="divDirectories"></div>
    </td>
    <td width="2" class="cssDivider"> </td>
```

```
        <td width="75%">
          <div class="cssFiles" id="divFiles"></div>
        </td>
      </tr>

    </table>

  </body>

</html>
```

Let's take it from the top, shall we? The first important bit we find is a JSP scriptlet section that gets the default system path separator character (\ on Windows, / on *nix). This will be needed later when paths are constructed, and this is a good, simple way to get that information. Note too that on a Windows-based system, the path separator happens to be the same character that Java, and JavaScript as well (not to mention most languages based on C syntax), uses as an escape sequence trigger. Therefore, if I had just used the value `File.separator` contains and tried to insert it into a string in JavaScript, it would have resulted in broken code that the browser couldn't parse, and thus a broken application. This is a pretty typical thing to have to do; but pointing it out was suggested by my technical reviewer, and he's a pretty smart dude, so here you have it!

Following that is a batch of JavaScript imports. We see the usual DWR imports of `engine.js` and `util.js`, in addition to the interface proxy for `FileSystemFunctions`. Following those are nine imports needed for the dhtmlx components, generally one for each, although the grid and the menubar require two each.

After the JavaScript files are imported, we need to get all the style sheets imported as well, so that comes next. Again, each dhtmlx component has its own style sheet, in addition to the main `styles.css` style sheet for Fileman.

That closes out the `<head>` of the document; now comes the `<body>`, and you'll immediately notice some work being done `onLoad`, namely a call to the `init()` method of the `Fileman` class instance named `fileman`, which we'll get a look at shortly. As you can imagine, `init()` handles all the initialization required on startup. Note that the file path separator we got in the scriplet section earlier is passed to this method. It will be stored there, so we have that value client side in the `fileman` instance.

You'll also see that `onResize` another call is made to the `onResize()` method of the `fileman` instance. This deals with resizing the file grid when the window is resized.

Now the markup that makes up the visual presentation of Fileman begins. First, we have `divFileEditor`, which is the `<div>` shown when the user wants to edit a file. It's not much more than a table with a `<textarea>` in the middle and two buttons, one for saving the edits (which calls the `saveFile()` method of `fileman`), and another to cancel editing (which simply hides `divFileEditor` again).

Next is `divFileUpload`, which is conceptually almost identical to `divFileEditor`, except that, of course, it is for uploading files to the server. Here we have ourselves a simple form that submits to `uploadFile.jsp` (if that seems a little weird, don't worry, I'll explain in the next section!). This form has a hidden field named `uploadDirectory`, which is the path of the currently selected directory, and is where the uploaded file will be written to on the server. There is also, naturally enough, an `<input>` element of type `file`, allowing the user to browse the local file system and select a file to upload. The Submit button is there to submit the form, and, like

with the file editor before it, there is a Cancel button, which simply hides `divFileUpload` again, if the user changes his or her mind.

Lastly is perhaps the most important, yet ironically most simplistic, portion of this JSP: the markup that organizes what the user sees. We see a single table that contains two rows. The first row contains a single cell that spans all three of the columns present in the table. Contained within this cell are two `<div>`s, one each for the menubar and toolbar. Notice that there's no content in them: the dhtmlx components create the necessary markup and insert it into these `<div>`s.

Below that row is the second row consisting of three cells. The first contains the `cssDirectories <div>`, which is where our directory tree will be rendered by the dhtmlx component. Following that is a small cell that is the divider between the directory tree and file grid. Lastly, the final cell in this row contains the `cssFiles <div>`, which is where the file grid will be rendered.

As you can see, this JSP is pretty darned simple, really just some basic markup that forms the structure of the application.

Oh yeah, what about that whole `uploadFile.jsp` thing? That was a little out of the ordinary, so let's use that as a good segue to the next section.

uploadFile.jsp

So, why did the form to upload a file submit to a JSP? That's a bit unusual, isn't it? Yes, it is, but not unprecedented either. The key point to remember is that a JSP is compiled into a servlet. So, after the initial compilation, what's the difference between a JSP and a servlet? Functionally, not much! So, in some cases, it saves time to simply create a JSP instead of a servlet that you then need to compile yourself, perhaps package into a WAR, configure in `web.xml`, and so on. While I typically wouldn't do this, based entirely on the idea that doing things with servlets, while more work, probably leads to a better-organized application (and a potentially flexible one, since you're in complete control rather than trusting the JSP compiler in the container), since we're only talking about two simple functions (uploading a file and downloading a file), I decided to skip the extra work involved and use JSPs instead. Just like using libraries, sometimes being lazy isn't a bad thing!

In Listing 6-8, you can see the full source for the `uploadFile.jsp`, the target of that file upload form submission we saw earlier.

Listing 6-8. *The* `uploadFile.jsp`*, Used by the User to Upload a File*

```
<%@ page language="java" import="java.util.*,java.io.*,➥
org.apache.commons.fileupload.*,➥
org.apache.commons.fileupload.disk.*,➥
org.apache.commons.fileupload.servlet.*,org.apache.commons.io.*" %>

<%
  /*
    This JSP is used to upload a file.  It used Commons FileUpload to parse the
    multipart request and save the file to the specified location.
  */
  FileItemFactory factory = new DiskFileItemFactory();
  ServletFileUpload upload = new ServletFileUpload(factory);
```

```
  List items = upload.parseRequest(request);
  Iterator it = items.iterator();
  HashMap params = new HashMap();
  FileItem file = null;
  // The elements of the request, either form field or the file itself.
  while (it.hasNext()) {
    FileItem item = (FileItem)it.next();
    if (item.isFormField()) {
      params.put(item.getFieldName(), item.getString());
    } else {
      file = item;
    }
  }
  // Get just the name portion of the uploaded file (it's the full path without
  // doing this work, which doesn't help us much).
  String incomingName = file.getName();
  String nakedFilename = FilenameUtils.getName(incomingName);
  try {
    InputStream is = file.getInputStream();
    OutputStream os = new FileOutputStream(
      (String)params.get("uploadDirectory") + File.separator + nakedFilename);
    int bytesRead = 0;
    byte[] buffer = new byte[8192];
    while((bytesRead = is.read(buffer, 0, 8192)) != -1) {
      os.write(buffer, 0, bytesRead);
    }
    is.close();
    os.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
%>

<html><head><title>File uploaded</title><script>
// Hide the file upload form in the parent so that when this pop-up is closed,
// the display is showing the file manager itself again.
opener.document.getElementById("divFileUpload").style.display = "none";
window.opener.fileman.toolbarButtonClick("tbRefresh", "");
</script></head><body>
File has been uploaded.  You may now close this window.
</body></html>
```

Here now, we get a look at Commons FileUpload in action. You'll find that the code, by and large, looks like the quick little example code I gave earlier in discussing Commons FileUpload. A `FileItemFactory` is instantiated and handed to a `ServletFileUpload` instance. Then, it's a simple matter of calling the `parse()` method of that instance, passing it the incoming `request` object, and we wind up with a `List` of `FileItem` objects, each of which is either a form field or an uploaded file. So, we iterate over that collection, and for each form field, we record

its name and value in the `params HashMap`. We do this because we need both the uploaded file and hidden form field telling us which directory to write the file to.

When that's done, we then use a handy little class called `FilenameUtils` found in the Commons IO package, and specifically the `getName()` method. This takes a fully qualified file name, say, `c:\temp\i_am_not_a_file.txt`, and returns just the file name, `i_am_not_a_file.txt`. (Oh, a wise guy, eh? Then again, it really might **not** be a file!) Once we have that, in addition to the value of that hidden form field (which should now be present in the `params HashMap` under the key `uploadDirectory`), we can write out the file. This is relatively straightforward Java Streams IO, so I'll leave those few lines of code to you to parse.

Once the file has been uploaded, since we're in a JSP after all, we can immediately output the response, which is just a simple HTML document. This document, upon loading, executes two lines of JavaScript: one that hides the file upload form in the parent, and another that simulates a click of the Refresh button to refresh the current directory, thereby showing the uploaded file. After that, a message is presented to the user to indicate completion of the upload and that he or she can close the pop-up. So, once the user closes this pop-up (which, in Firefox or IE7, depending on your settings, could be a different tab), he or she will again be back in Fileman, and the display will be returned to normal, with the directory tree and file grid visible and updated.

downloadFile.jsp

Like uploading a file, downloading a file takes the lazy approach and uses a JSP. This one is even simpler than the `uploadFile.jsp`, as shown in Listing 6-9.

Listing 6-9. *The* `downloadFile.jsp`*, Executed When the User Wants to Download a File*

```
<%@ page language="java" import="java.io.*" %>

<%
  /*
    This JSP is used to download a file.  It accepts the path and name of the
    file to download, and streams it back.  Most browsers should try and
    recognize the file type; otherwise it should be downloaded as a stream of
    bytes only, which should always be fine.
  */
  String filePath = request.getParameter("p");
  String fileName = request.getParameter("n");
  response.setContentType("application/unknown");
  response.addHeader("Content-Disposition",
    "attachment; filename=" + fileName);
  try {
    File f = new File(filePath + File.separator + fileName);
    int fSize = (int)f.length();
    FileInputStream fis = new FileInputStream(f);
    PrintWriter pw =  response.getWriter();
    int c = -1;
```

```
    while ((c = fis.read()) != -1) {
      pw.print((char)c);
    }
    fis.close();
    pw.flush();
    pw = null;
  } catch (Exception e) {
    e.printStackTrace();
  }
%>
```

Two parameters are passed to this JSP, `p` and `n`. The parameter `p` names the path to the file to download, and `n` names the file itself. With those two pieces of information, it's a simple matter for Streams IO to write out the file to the response. We have to be sure to set the `Content-Disposition` header to indicate a download though, and you can see that here. We also set the content type to `application/unknown`, so the browser shouldn't try and interpret the data, just pop up a Save File dialog box and let the user choose a download location.

Fileman.js

Now we come to the really meaty portion of the client-side code, namely the `Fileman` JavaScript class. Because this `Fileman.js` file weighs in at nearly a thousand lines of code, I didn't think 7–8 pages of code here would make much sense, so instead I'll pull out section by section for you to look at as we discuss it.

We begin our dissection of this class with a look at the private data members, as enumerated in Table 6-2. The descriptions included in the table should tell you all you need to know about each.

Table 6-2. *The Private Members Found in the* `Fileman` *Class*

Private Member	Description
`directoryTree`	Reference to the directory tree component.
`fileGrid`	Reference to the file grid component.
`Menubar`	Reference to the menubar component.
`Toolbar`	Reference to the toolbar component.
`pathSeparator`	The system-specific path separator character for the host system.
`clipboardPath`	The full path of the object (file or directory) on the clipboard.
`clipboardName`	The name of the object (file or directory) on the clipboard.
`clipboardType`	The type of the object (file or directory) on the clipboard. Value is either null if none, `file`, or `directory`.
`clipboardOperation`	The operation active for the item currently on the clipboard. Value is either `null` if none, `copy`, or `cut`.
`fileBeingEdited`	The full path to the file currently being edited, if any.

init() Method

The first of the many methods we encounter in the `Fileman` class is the `init()` method that we saw called `onLoad` of the `index.jsp` file earlier. This is a fairly lengthy method, so I'm going to discuss it in parts, beginning with the opening of the method:

```
this.init = function(inPathSeparator) {

  // Record the path separator for the host system for later.
  pathSeparator = inPathSeparator;
```

Recall that the system path separator character is recorded in the JSP scriplet that starts `index.jsp`, and that value is then passed to this `init()` method. So, the first thing done is to store that character in the `pathSeparator` field for use later.

The next task `init()` performs is to build the directory tree, as follows:

```
// Create and configure the directory tree component.
directoryTree = new dhtmlXTreeObject("divDirectories", "100%", "100%", 0);
directoryTree.setImagePath("img/");
directoryTree.attachEvent("onClick", fileman.directoryClicked);
directoryTree.attachEvent("onDblClick", function() { } );
directoryTree.attachEvent("onOpenStart", fileman.directoryExpanded);
directoryTree.setStdImages("folderClosed.gif", "folderOpen.gif",
  "folderClosed.gif");
directoryTree.insertNewChild(0, "0_dummy", "Loading...",
  0, "blank.gif", 0, 0, 0, 0);
```

With the quick example code shown earlier when we first looked at the dhtmlx components, this shouldn't be anything too surprising. After the `dhtmlXTreeObject` is instantiated, a reference to it is stored in the `directoryTree` field. We then tell the object what the path to the images is, so they can be used to render the tree properly. We then connect an `onClick` event handler to the tree such that the `directoryClicked()` method of the `Fileman` class will be called when any node in the tree is clicked (we'll be seeing all these event handlers shortly). Likewise, the `directoryExpanded()` method will be called when any node in the tree is expanded (the `dhtmlXTreeObject` class offers a number of events, as do all the dhtmlx components . . . see the corresponding component documentation for full details of what's available). We also tell the component what the standard images are for the nodes when they have no children, when they are expanded, and when they aren't expanded. Finally, a single node is inserted into the tree that says "Loading". This will be removed as soon as the root file systems are populated, which is coming up soon.

Note that the double-click event is handled too, but just an empty function is passed to it. If this event isn't handled in some way, all manner of nastiness ensues . . . try removing that line and double-click a directory, and prepare for a mess!

The next code to look at is the code for building the file grid:

```
// Create and configure the file grid component.
fileGrid = new dhtmlXGridObject();
fileGrid.setImagePath("img/");
fileGrid.setHeader("Name,Size,Type,Modified");
fileGrid.setColAlign("left,right,left,right")
fileGrid.setColTypes("ed,ro,ro,ro");
fileGrid.setInitWidthsP("40,20,20,20");
fileGrid.setColSorting("str,str,str,str");
var gridHeight = getContentAreaHeight() - 70;
fileGrid.enableAutoHeight(true, gridHeight, false);
fileGrid.attachToObject(dwr.util.byId("divFiles"));
fileGrid.setOnEditCellHandler(fileman.filenameChanged)
fileGrid.init();
```

After instantiating the object, we again tell it where its image resources can be found. Then, the `setHeader()` method is called to create the four columns we see. The next four function calls tell the grid how text in the columns should be aligned (`setColAlign()`), whether each column is editable (`ed`) or read-only (`ro`) (`setColTypes()`), the initial widths of the columns in pixels (`setInitWidthsP()`), and what type of data is present in the columns for sorting purposes (`setColSorting()`), which for our purposes are all strings. The next two lines are responsible for setting the height of the grid so that it fills the `<div>` it's housed in. The `getContentAreaHeight()` is another method of the `Fileman` class that returns the maximum size of the browser content area. That value is taken, then 70 subtracted from it to account for the menubar and toolbar, and then that value is sent to `enableAutoHeight()`. Finally, we call the `attachToObject()` method of `dhtmlXGridObject` and pass it a reference to the `divFiles` `<div>` that will contain the grid. We use DWR's `dwr.util.byId()` function to get that reference. Next, we attach an event handler that will fire when any cell in the table is edited (which applies only to the cells in the name column for this application), which is `filenameChanged()` in the `Fileman` class. Finally, the `init()` method of the grid object is called, and we have ourselves a file grid, all set up and ready to go.

The next bit of code deals with building the toolbar, and it is as follows:

```
// Create and configure toolbar.
toolbar = new dhtmlXToolbarObject(
  dwr.util.byId("divToolbar"), "100%", "20px", "");
toolbar.setOnClickHandler(fileman.toolbarButtonClick);
var tbUpOneLevel = new dhtmlXImageButtonObject(
  "img/icoUpOneLevel.gif", "38px", "32px", null, "tbUpOneLevel",
  "Up one level (Go to parent directory)", null,
  disableImage="img/icoUpOneLevel.gif");
var tbRefresh = new dhtmlXImageButtonObject(
  "img/icoRefresh.gif", "38px", "32px", null, "tbRefresh",
  "Refresh current directory", null, disableImage="img/icoRefresh.gif");
var tbNewFile = new dhtmlXImageButtonObject(
  "img/icoNewFile.gif", "38px", "32px", null, "tbNewFile",
  "Create a new file", null, disableImage="img/icoNewFile.gif");
var tbNewDirectory = new dhtmlXImageButtonObject(
  "img/icoNewDirectory.gif", "38px", "32px", null, "tbNewDirectory",
```

```
  "Create a new directory", null, disableImage="img/icoNewDirectory.gif");
var tbEditFile = new dhtmlXImageButtonObject(
  "img/icoEditFile.gif", "38px", "32px", null, "tbEditFile",
  "Edit selected file", null, disableImage="img/icoEditFile.gif");
var tbDelete = new dhtmlXImageButtonObject(
  "img/icoDelete.gif", "38px", "32px", null, "tbDelete",
  "Delete selected file/directory", null,
  disableImage="img/icoDelete.gif");
var tbCopy = new dhtmlXImageButtonObject(
  "img/icoCopy.gif", "38px", "32px", null, "tbCopy",
  "Copy selected file/directory", null,
  disableImage="img/icoCopy.gif");
var tbCut = new dhtmlXImageButtonObject(
  "img/icoCut.gif", "38px", "32px", null, "tbCut",
  "Cut selected file/directory", null,
  disableImage="img/icoCut.gif");
var tbPaste = new dhtmlXImageButtonObject(
  "img/icoPaste.gif", "38px", "32px", null, "tbPaste",
  "Paste file/directory currently on clipboard", null,
  disableImage="img/icoPaste.gif");
var tbUpload = new dhtmlXImageButtonObject(
  "img/icoUpload.gif", "38px", "32px", null, "tbUpload",
  "Upload file", null, disableImage="img/icoUpload.gif");
var tbDownload = new dhtmlXImageButtonObject(
  "img/icoDownload.gif", "38px", "32px", null, "tbDownload",
  "Download file", null, disableImage="img/icoDownload.gif");
toolbar.addItem(tbUpOneLevel);
toolbar.addItem(tbRefresh);
toolbar.addItem(new dhtmlXToolbarDividerXObject("divider1"));
toolbar.addItem(tbNewFile);
toolbar.addItem(tbNewDirectory);
toolbar.addItem(new dhtmlXToolbarDividerXObject("divider2"));
toolbar.addItem(tbEditFile);
toolbar.addItem(new dhtmlXToolbarDividerXObject("divider3"));
toolbar.addItem(tbDelete);
toolbar.addItem(new dhtmlXToolbarDividerXObject("divider4"));
toolbar.addItem(tbCopy);
toolbar.addItem(tbCut);
toolbar.addItem(tbPaste);
toolbar.addItem(new dhtmlXToolbarDividerXObject("divider5"));
toolbar.addItem(tbUpload);
toolbar.addItem(tbDownload);
toolbar.showBar();
```

This is for the most part just an extension of the example code you saw earlier, so I won't say too much about it. The only new thing is the use of the `dhtmlXToolbarDividerXObject`, which is a divider bar on the menu. It's treated basically like any other item on the toolbar; you create an instance of it and use the `addItem()` method of the toolbar to add it.

Next up is the menubar construction code:

```
// Create and configure the menu.
menubar = new dhtmlXMenuBarObject("divMenu", "100%", 22, "DWR File Manager");
menubar.setGfxPath("img/");
menubar.setOnClickHandler(fileman.menubarButtonClick);
var item = null;
var subMenu = null;
// File menu.
item = new dhtmlXMenuItemObject("mnuFile", "File", "");
menubar.addItem(menubar, item);
subMenu = new dhtmlXMenuBarPanelObject(menubar, item, false, 120, true);
item = new dhtmlXMenuItemObject("mnuDelete", "Delete", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuDividerYObject("mnuFileDivider1");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuNewFile", "New File", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuNewDirectory", "New Directory", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuDividerYObject("mnuFileDivider2");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuUpload", "Upload", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuDownload", "Download", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuDividerYObject("mnuFileDivider3");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuExit", "Exit", "");
menubar.addItem(subMenu, item);
// Edit menu.
item = new dhtmlXMenuItemObject("mnuEdit", "Edit", "");
menubar.addItem(menubar, item);
subMenu = new dhtmlXMenuBarPanelObject(menubar, item, false, 120, true);
item = new dhtmlXMenuItemObject("mnuEditFile", "Edit File", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuDividerYObject("mnuEditDivider1");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuCopy", "Copy", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuCut", "Cut", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuPaste", "Paste", "");
menubar.addItem(subMenu, item);
// Tools menu.
item = new dhtmlXMenuItemObject("mnuTools", "Tools", "");
menubar.addItem(menubar, item);
subMenu = new dhtmlXMenuBarPanelObject(menubar, item, false, 120, true);
```

```
item = new dhtmlXMenuItemObject("mnuPrintDirectoryContents",
  "Print Directory Contents", "");
menubar.addItem(subMenu, item);
// Help menu.
item = new dhtmlXMenuItemObject("mnuHelp", "Help", "");
menubar.addItem(menubar, item);
subMenu = new dhtmlXMenuBarPanelObject(menubar, item, false, 120, true);
item = new dhtmlXMenuItemObject("mnuUsingFileman", "Using Fileman", "");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuDividerYObject("mnuHelpDivider1");
menubar.addItem(subMenu, item);
item = new dhtmlXMenuItemObject("mnuAbout", "About...", "");
menubar.addItem(subMenu, item);
```

I'm sure you're probably seeing a pattern by now: most of the dhtmlx components share a similar general interface when it comes to constructing them programmatically. Once an instance of the `dhtmlXMenubarObject` class is created, it is fed the path to its images, and a click handler is attached to it that will be called when any menu item is clicked. After that, it's a simple matter of creating `dhtmlXMenuItemObject` instances and adding them to the menu. These are the top-level menus: File, Edit, Tools, and Help. To each we add a `dhtmlXMenuBarPanelObject`, which represents a submenu. Finally, to each submenu, we add `dhtmlXItemObject` instances for each menu item and `dhtmlXDividerYObjecct` instances where we want dividers to appear. Hopefully this code is mostly self-evident. Interestingly, note that there is no need to call a function at the end to show the menubar like with the toolbar; it's automatically on the screen when it's instantiated (remember that the first parameter passed to the constructor of the `dhtmlXMenubarObject` is the `<div>` to insert the menu into).

If it seems like I've glossed over the dhtmlx code a little bit, it's because I have, and that's no accident! Although I've tried to give you enough to understand it, this is after all a book about DWR, so it wouldn't make much sense to go into every last detail of something else. Since the documentation for the components is quite good, you shouldn't have any problem finding answers to any lingering questions you may have just by looking at the API documentation. I hope you don't view this as too big a cop-out, but if you really want more in-depth coverage, feel free to write Apress and tell them you want me to write a book on dhtmlx (and perhaps other component libraries) . . . I'm a lot like Kevin Smith[4] in that we'll both do just about anything for money!

With the dhtmlx components fully constructed, it's time to turn our attention to the DWR initialization code, which really amounts to a single line of code:

---

4. Kevin Smith is the well-known, famously self-deprecating director of such cinema classics as *Clerks, Jay and Silent Bob Strike Back, Mall Rats*, and <sarcasm>*Jersey Girls*</sarcasm>. He's also done a number of speaking engagements and released them on DVD (more evidence of his ability to do anything for money!) that are highly entertaining. See `http://imdb.com/find?s=all&q=kevin+smith` for all the details you could want.

```
// Configure exception handler for DWR.
dwr.engine.setErrorHandler(fileman.exceptionHandler);
```

This sets up a global exception handler that will be called when any sort of exception occurs during a remote call. You'll see how this is used very soon.

The final initialization task required when Fileman starts up is to retrieve a list of file system roots and populate the directory tree with them. That gives the user a proper starting place to begin work. The code that accomplishes that is a remote call, as shown here:

```
// List file system roots.
FileSystemFunctions.listRoots(
  { callback : function(inResp) {
      directoryTree.deleteItem("0_dummy");
      // Iterate over the collection of DirectoryVO objects returned.
      for (var i = 0; i < inResp.length; i++) {
        var nodeID = i + inResp[i].name;
        // Add file system root to tree.
        directoryTree.insertNewChild(0, nodeID, inResp[i].name, 0,
          "folderClosed.gif", "folderOpen.gif", "folderClosed.gif", 0, 0);
        // Set path of root on node for later.
        directoryTree.setUserData(nodeID, "path", inResp[i].path);
        // If the root has subdirectories, add the "Loading" dummy node
        // so that when they expand the node they'll see this until the
        // contents load.
        if (inResp[i].hasChildren) {
          directoryTree.insertNewChild(nodeID, nodeID + "_dummy",
            "Loading...", 0, "blank.gif", 0, 0, 0, 0);
          directoryTree.closeItem(nodeID);
        }
      }
    }
  }
);
```

So, we're calling on the `listRoots()` method of the `FileSystemFunctions` class on the server. This is the only class that is remoted, remember, so you'll be seeing nothing but it in the DWR calls. Once the response comes back, the first thing that's done is to remove the item we added to the tree when it was created. Remember that this has the text "Loading...", so what the user sees is a message in the tree saying the contents are loading, and then when the contents are returned to the client, that message is removed before the contents are added. Once that's done, we iterate over the collection of `DirectoryVO` objects that was returned by the remote call. For each, we construct a unique node ID for the node in the tree, and then we use the `insertNewChild()` method of the tree to add it. This call specifies the node ID, the text to display, and the images to use. Also to each node we add a custom piece of data, which is a feature the tree supports. This information is the path of the file system, such as `C:\` for a typical Windows system, for example. Lastly, for any node that has children (which is a flag set on the `DirectoryVO`), we add a similar "Loading" node to it like we did to the initial tree. This will have the same effect: when the user expands a node, he or she will see the loading message

while the child directories are retrieved. Finally, we have to explicitly close the node if it has children because by default the tree wants to expand that node, and that's not what we want here.

exceptionHandler() Method

With the `init()` method out of the way, we'll now start looking over all the methods that handle various events the application can encounter, beginning with that global exception handler we just saw configured for all DWR calls:

```
this.exceptionHandler = function(inMessage) {

  alert(inMessage);

} // End exceptionHandler().
```

Well, that really couldn't be simpler! The message of the exception is simply displayed in an `alert()` pop-up, and that's that. The server is therefore responsible for giving the UI a useful message and not just some error code or something like that, so we'll be sure to do that when we hit the `FileSystemFunctions` class.

onResize() Method

Next is the `onResize()` method, which as the name implies (and as we already in fact saw when we looked at `index.jsp`) is called when the window is resized:

```
this.onResize = function() {

  var gridHeight = getContentAreaHeight() - 70;
  fileGrid.enableAutoHeight(true, gridHeight, false);

} // End onResize().
```

Here, you can see the same code you saw in `init()`, the goal of which is to resize the file grid to fill the content area of the page.

getContentAreaHeight() Method

In both the `init()` method and the `onResize()` method, you saw a call to `getContentAreaHeight()`, and now we're confronted with the code of that method:

```
getContentAreaHeight = function() {

  var myHeight = 0;
  if (typeof(window.innerHeight) == "number") {
    // Non-IE
    myHeight = window.innerHeight;
  } else if (document.documentElement &&
    document.documentElement.clientHeight) {
    // IE 6+ in "standards compliant mode".
    myHeight = document.documentElement.clientHeight;
```

```
  } else if (document.body && document.body.clientHeight) {
    // IE 4 compatible.
    myHeight = document.body.clientHeight;
  }
  return myHeight;

} // End getContentAreaHeight().
```

In short, this method deals with all the cross-browser issues and at the end gets a pretty accurate number that represents the pixel height of the content area of the browser, that is, the size of the area the document is displayed in. Because various browsers provide this information via differing properties of the `window` and/or `document` object, and worse still, IE gives different values depending on whether the document is in quirks mode or standards-compliant (a.k.a. strict) mode, there is some logic that needs to be done there to get the proper value. But, in the end, this code gets it, and returns it, which is very handy indeed.

menubarButtonClick() Method

When any item in the menu is clicked, the `menubarButtonClick()` method is called, because it was attached as the event handler to the menubar in `init()`.

```
this.menubarButtonClick = function(inItemID, inItemValue) {

  // Get the ID of the selected directory and file, if any.
  var selectedDirectory = directoryTree.getSelectedItemId();
  var selectedItem = fileGrid.getSelectedId();

  // Branch based on button clicked.
  if (inItemID == "mnuDelete") {
    // Delete menu item clicked.
    fileman.doDelete(selectedDirectory, selectedItem);
  } else if (inItemID == "mnuNewFile") {
    // Create new file menu item clicked.
    fileman.doNewFile(selectedDirectory);
  } else if (inItemID == "mnuNewDirectory") {
    // Create new directory menu item clicked.
    fileman.doNewDirectory(selectedDirectory);
  } else if (inItemID == "mnuEditFile") {
    // Edit File menu item clicked.
    fileman.doEditFile(selectedDirectory, selectedItem);
  } else if (inItemID == "mnuCopy") {
    // Copy menu item clicked.
    fileman.doCopyCut(selectedDirectory, selectedItem, "copy");
  } else if (inItemID == "mnuCut") {
    // Cut menu item clicked.
    fileman.doCopyCut(selectedDirectory, selectedItem, "cut");
  } else if (inItemID == "mnuPaste") {
    // Paste menu item clicked.
    fileman.doPaste(selectedDirectory);
```

```
  } else if (inItemID == "mnuDownload") {
    // Download menu item clicked.
    fileman.doDownload(selectedDirectory, selectedItem);
  } else if (inItemID == "mnuUpload") {
    // Upload menu item clicked.
    fileman.doUpload(selectedDirectory);
  } else if (inItemID == "mnuPrintDirectoryContents") {
    // Print Directory Contents menu item clicked.
    fileman.doPrintDirectoryContents(selectedDirectory);
  } else if (inItemID == "mnuUsingFileman") {
    // Using Fileman menu item clicked.
    window.open("using.htm");
  } else if (inItemID == "mnuAbout") {
    // About menu item clicked.
    fileman.doAbout();
  }

} // End menubarButtonClick().
```

It begins by grabbing the node ID of the currently selected directory in the tree, and the row ID of the currently selected file in the file grid, if any. This information is passed to most of the functions that are called. Take note that it's not the actual path or file name that's captured here, it's node and row IDs in the tree and grid components. This is important because, as you probably suspect at this point, there's going to be code later to get the actual path and file name as required, but that's not done here.

After that, it's nothing but a giant `if...else if` block where each `if` check is against a menu item ID (as seen in the menu creation code in `init()`). The appropriate method is called in each case, passing the pertinent information (directory node ID, file row ID, both, or neither).

The only exception to this model is the Using Fileman menu item. Since this is a single line of code, there's no `doUsing()` method anywhere. Instead, a new pop-up window is opened, and the `using.htm` file is loaded into it, and that's mission accomplished (and I mean **really** accomplished, not in the Bush Administration sort of accomplished way!).

toolbarButtonClick() Method

The toolbar's click event handler, `toolbarButtonClick()`, is basically the same as the `menubarButtonClick()` method, and so I haven't printed it out here. Have a look though, and you'll see it's again just a giant `if...else if` block, and nothing more. It might have been possible to somehow use the same method for both these events (it in fact may have worked to give the toolbar buttons and menu items the same IDs, and therefore the same code should have worked), but I decided against that simply to keep this more extensible. I figured this way any differences in the menu and toolbar (because there could be features exposed by one but not the other) could be dealt with easily. This flexibility is paid for by a bit of redundant code, but as simplistic as that code is, I don't think it's an egregious design choice.

getFullPath() Method

The `getFullPath()` method is a quick little utility method that is used in a couple of places throughout the rest of the `Fileman` code:

```
this.getFullPath = function(inSelectedDirectory, inSelectedItem) {

  var directoryPath = directoryTree.getUserData(inSelectedDirectory, "path");
  var fileName = fileGrid.getUserData(inSelectedItem, "name");
  return directoryPath + pathSeparator + fileName;

} // End getFullPath().
```

Its job, given a node ID of a directory and a row ID of a file, is to return a full path to that file (or directory). To do this, we take the node ID passed in and use the `getUserData()` method of the dhtmlxTree component to get the `path` value we stored there when the tree was populated (we've only seen this for the root file systems, but as you'll see later, it's the case for all directories added to the tree). We then take the row ID passed in and use the `getUserData()` method of the dhtmlxGrid component to get the user data value `name`, which is just the file name (or directory name) of the selected item in the grid. With those two pieces of information, plus that system file path separator character we stored during initialization, we can form a full path to the file or directory and return it.

directoryExpanded() Method

Now we come to a method with a bit more oomph to it, the `directoryExpanded()` method, which is called whenever a node in the directory tree is expanded. Its job is to retrieve a list of directories that are children of the expanded directory and populate the tree with them.

```
this.directoryExpanded = function(inNodeID, inCurrentState) {

  // If the first child is the "Loading" dummy node, we need to load
  // the contents of subdirectories of the directory.
  var firstNode = directoryTree.getChildItemIdByIndex(inNodeID, 0);
  if (firstNode.indexOf("_dummy") != -1) {
    var path = directoryTree.getUserData(inNodeID, "path");
    FileSystemFunctions.listDirectories(path,
      { callback : function(inResp) {
          // Iterate over the collection of DirectoryVO objects returned.
          for (var i = 0; i < inResp.length; i++) {
            var nodeID = inNodeID + i + inResp[i].name;
            // Add the subdirectory to its parent.
            directoryTree.insertNewChild(inNodeID, nodeID,
              inResp[i].name, 0, "folderClosed.gif", "folderOpen.gif",
              "folderClosed.gif", 0, 0);
            // Set path of root on node for later.
            directoryTree.setUserData(nodeID, "path", inResp[i].path);
            // If the subdirectory itself has subdirectories, add the
            // "Loading" dummy node so that when they expand the node they'll
            // see this until the contents load.
            if (inResp[i].hasChildren) {
              directoryTree.insertNewChild(nodeID, nodeID + "_dummy",
                "Loading...", 0, "blank.gif", 0, 0, 0, 0);
              directoryTree.closeItem(nodeID);
```

```
            }
          }
          // Delete the "Loading" node.
          directoryTree.deleteItem(inNodeID + "_dummy");
        }
      }
    );
  }
  return true;

} // End directoryExpanded().
```

The first thing that's done is a check: is the very first child node of the selected node the "Loading" node? Only in this case do we have work to do because it means we haven't yet populated this branch of the tree.

Assuming we have work to do, that work begins by getting the path of the expanded directory. That information is passed as a parameter to the `listDirectories()` method of the `FileSystemFunctions` remote class. This method will get the subdirectories, starting at the directory specified by the path parameter.

Upon return from this method, we begin to iterate over the collection of `DirectoryVO`s returned, one for each child directory (if any). For each, we construct a unique node ID to represent the directory in the tree, and we then use the `insertNewChild()` method of the dhtmlxTree component to add a node for the directory. The `setUserData()` method is also used to store the actual path of the directory on the node for later use.

Next, we check the value of the `hasChildren` flag on the `DirectoryVO`. If it's `true`, that means the subdirectory itself has child directories, in which case we add a "Loading" node to it so that when the user expands it, it works just like every other directory, initially showing "Loading..." while the subdirectories for it are retrieved. Finally, we call the `closeItem()` method on the tree component, passing it the ID of the node we just added. This is necessary because the node would be expanded now due to it having a child node, which isn't what we want here.

The last step involved is to delete the "Loading" node in the selected directory. Once that's done, the callback is finished, and the tree is fully updated.

directoryClicked() Method

The `directoryClicked()` method is the next method you encounter as you walk through the `Fileman` class, and its job is to display the contents of a directory when the user clicks it in the directory tree.

```
this.directoryClicked = function(inNodeID) {

  // Make sure a directory is selected.
  if (inNodeID == null || inNodeID == "") {
    return;
  }
```

```
  // Get path of clicked directory.
  var path = directoryTree.getUserData(inNodeID, "path");

  // Clear grid and put in loading message.
  fileGrid.clearAll();
  fileGrid.addRow("rowLoading", "Loading...,,,,");
  FileSystemFunctions.listFiles(path,
    { callback : function(inResp) {
        fileGrid.deleteRow("rowLoading");
        // Iterate over the collection of FileVO objects returned.
        for (var i = 0; i < inResp.length; i++) {
          // For each, add a row to the grid.
          fileGrid.addRow(i + inResp[i].name, inResp[i].name + "," +
            inResp[i].size + "," + inResp[i].type + "," +
            inResp[i].modified);
          fileGrid.setUserData(i + inResp[i].name, "name", inResp[i].name);
        }
      }
    }
  );

} // End directoryClicked().
```

To accomplish the feat, the code begins by ensuring that a directory is selected. This is probably not necessary, as there likely is no possibility of this ever occurring, but for the sake of consistent code, I decided to leave the check in. Besides, as new capabilities are added in the future, it may be that this becomes a possible situation anyway, so it'd be nice if the code was already there to stop the application from breaking.

Anyway, after that check is a call to `getUserData()` on the tree component to get the path of the clicked directory. With that information ready, we can proceed to call on the server.

Before that call though, the contents of the grid, if any, are cleared via a simple call to the grid's `clearAll()` method. Also, we want to have a nice "Loading..." message while the contents are retrieved, so the `addRow()` grid method is used. You may find the text there a little funky: "Loading...,,,". That's actually not a typo: remember that we have four columns in the file grid, but the loading message is only being put in the first column. We still need to account for the other three columns though, and that's where those commas come in: imagine a value in between them, and that would be the text put into the other columns in the row, but since we want no content in them right now, just a list of commas makes everything work.

Finally, DWR is used to call the `listFiles()` method of the `FileSystemFunctions` class, passing it the path of the clicked directory. Upon returning, we delete our "Loading..." row and prepare to build the real contents of the grid.

The return from this method is an array of `FileVO` objects, so now it's time to iterate over that array. For each file (or directory, remember it could be either), we add a new row to the grid, and also set the name of the item on the row for later retrieval using the `setUserData()` method.

Would you care to guess what will happen if the file name returned has a comma in it? I'll save you the time: nothing good! The columns will be pushed down, and things will simply not be lined up or work properly. Commas are bad, m'kay? Seriously though, consider this one possible enhancement for you to do later: there are alternative ways to load the grid, and some exploration of the documentation should get you close to an answer in no time.

filenameChanged() Method

The user can rename files and directories any time by double-clicking the name column of the item he or she wishes to edit or clicking F2 when the cell is highlighted. When this happens, as we saw in the `init()` method, the `filenameChanged()` method will be called. More accurately, it will be called multiple times during the "lifetime" of the edit. This code, which handles the renaming operation, is as follows:

```
this.filenameChanged = function(inStage, inRowID, inCellIndex, inNewValue,
  inOldValue) {

  if (inStage == 2) {

    // Get the ID of the selected directory and file, if any.
    var selectedDirectory = directoryTree.getSelectedItemId();

    // Get fully qualified name of selected file.
    var directoryPath = directoryTree.getUserData(selectedDirectory, "path");
    var oldFullPath = directoryPath + pathSeparator + inOldValue;
    var newFullPath = directoryPath + pathSeparator + inNewValue;
    FileSystemFunctions.renameFile(oldFullPath, newFullPath,
      { callback : function(inResp) {
          // Refresh both the directory file list and the directory tree.
          fileman.directoryClicked(selectedDirectory);
          directoryTree.deleteChildItems(selectedDirectory);
          // Add the "Loading" node back in before refreshing tree.
          directoryTree.insertNewChild(selectedDirectory,
            selectedDirectory + "_dummy", "Loading...", 0, "blank.gif",
            0, 0, 0, 0);
          fileman.directoryExpanded(selectedDirectory);
        }
      }
    );
    return true;

  }

} // End filenameChanged().
```

The first thing we need to do is check that we're in stage 2. The reason for doing so is because this method will be called a total of three times for an edit: once before the editing begins, once when the cell editor is opened, and once when it closes. For our purposes here, we only care about that last stage, which is stage 2 (the stages go in the order stated here and are 0-based).

Once we confirm it's time to actually do the rename, we then go ahead and begin by getting the node of the selected directory in the tree. We then get the path of that directory by looking up the `path` user data attribute via the `getUserData()` method of the tree component. Next, we construct the full path to the item as it was originally named. As you can see, one of the parameters to this event-handler function is the old value of the edited cell. We do the same for the new name of the item, so the end result is two variables, `oldFullPath` and `newFullPath`, which represent the fully qualified path and name of the file or directory being renamed, both before and after the edit.

After that, it's a simple matter of a DWR call to the `renameFile()` method of the `FileSystemFunctions` class, passing it those two paths. The callback handler now has the chore of updating the display. To do so, the `directoryClicked()` method is called, which will update the file grid display with the new name. For directories, this also affects the directory tree, which requires us to delete the node corresponding to the selected directory, add a "Loading" node to it (so it now looks like it did when first created), and then call the `directoryExpanded()` method, which reacquires the subdirectories and updates the tree.

doCopyCut() Method

Whether copying or cutting a file or directory, it's basically the same operation: store the details of the selected item to the clipboard so that when paste is selected, we can perform the appropriate operation. Both copy and cut call this `doCopyCut()` method:

```
this.doCopyCut = function(inSelectedDirectory, inSelectedItem, inOperation) {

  // Only do something if both a directory and an item in the grid are
  // selected.
  if (inSelectedDirectory == null || inSelectedItem == null ||
    inSelectedDirectory == "" || inSelectedItem == "") {
    alert("Please select a file or directory to " + inOperation);
    return;
  }

  // Store the operation and full path to the item selected.
  this.clipboardOperation = inOperation;
  this.clipboardPath = directoryTree.getUserData(inSelectedDirectory, "path");
  this.clipboardName = fileGrid.getUserData(inSelectedItem, "name");
  var itemType = fileGrid.cells(inSelectedItem, 2).getValue();
  if (itemType == "Directory") {
    this.clipboardType = "directory";
  } else {
    this.clipboardType = "file";
  }

} // End doCopyCut().
```

First, we check to be sure both a directory in the tree and a file (or directory) in the grid is selected, and we abort with an alert message if that's not the case. Then, we store which operation is being performed as passed in via the `inOperation` parameter, as well as the tree node ID of the selected directory and row ID of the selected file or directory in the grid. Finally, we get the value of the third column (cell) in the grid (the cells are 0-based, so name is 0, size is 1, and type is 2) and set the `clipboardType` according to that value. At this point, we are ready to paste the copied/cut item when the user wants to.

doPaste() Method

The `doPaste()` method complements the `doCopyCut()` method, the ying to the other's yang, if you will. Once a file or directory is copied or cut to the clipboard, the `doPaste()` method can complete the operation as requested.

```
this.doPaste = function(inSelectedDirectory) {

  // Only do something if both a directory is selected and there is an
  // object on the clipboard.
  if (inSelectedDirectory == null || this.clipboardPath == null ||
    this.clipboardName == null) {
    alert("Nothing to paste");
    return;
  }

  // Call the copy/move function.
  var destinationPath =
    directoryTree.getUserData(inSelectedDirectory, "path");
  FileSystemFunctions.copyMoveFile(this.clipboardPath, this.clipboardName,
    destinationPath, this.clipboardOperation, this.clipboardType,
    { callback : function(inResp) {
        // Clear all clipboard variables.
        fileman.clipboardPath = null;
        fileman.clipboardName = null;
        fileman.clipboardOperation = null;
        fileman.clipboardType = null;
        // Update the destination directory.
        fileman.directoryClicked(inSelectedDirectory);
      }
    }
  );

} // End doPast().
```

Once the code ensures a directory is currently selected (otherwise, how would we know where to put the item on the clipboard?) and also that something is currently on the clipboard, it proceeds to call the `copyMoveFile()` method of the `FileSystemFunctions` class, passing it six pieces of information:

- The path of the item being copied or cut
- The name of the item being copied or cut
- The path of the directory to place the item
- Whether the file is being copied or cut
- Whether the item on the clipboard is a file or a directory

This is all the information the server needs to perform the operation. When the callback is executed, all that's left to do is clean up the variables defining the clipboard so that it's empty and prepared to accept another item, and also a call to `directoryClicked()` is required so that the directory tree is updated appropriately, since the item copied or cut may have been a directory.

If you'd like Fileman to work a little more like Windows Explorer, you might consider getting rid of the four lines of code that clear the clipboard. This will allow you to paste a copied item as many times as you'd like. I like the idea of not being able to accidentally create numerous copies of something just because my hand twitches a bit hitting the Paste button, but you may not be quite as spastic as me!

---

doDelete() Method

The `doDelete()` method is called to delete the currently selected file or directory in the file grid. Its code is as follows:

```
this.doDelete = function(inSelectedDirectory, inSelectedItem) {

  // Only do something if both a directory and an item in the grid are
  // selected.
  if (inSelectedDirectory == null || inSelectedItem == null ||
    inSelectedDirectory == "" || inSelectedItem == "") {
    alert("Please select a file or directory to delete");
    return;
  }

  // Get fully qualified name of selected item.
  var fullPath = fileman.getFullPath(inSelectedDirectory, inSelectedItem);

  // Confirm deletion, then make the call to do it if confirmed.
  if (confirm("Are you sure you want to delete the following " +
    "file/directory?\n\n" + fullPath)) {
    // OK, user confirmed deletion, kill the file!
    FileSystemFunctions.deleteFile(fullPath,
      { callback : function(inResp) {
          // Refresh both the directory file list and the directory tree.
          fileman.directoryClicked(inSelectedDirectory);
```

```
          directoryTree.deleteChildItems(inSelectedDirectory);
          // Add the "Loading" node back in before refreshing tree.
          directoryTree.insertNewChild(inSelectedDirectory,
            inSelectedDirectory + "_dummy", "Loading...", 0, "blank.gif",
            0, 0, 0, 0);
          fileman.directoryExpanded(inSelectedDirectory);
        }
      }
    );
  }

} // End doDelete().
```

It begins with a check to be sure a directory in the tree and file/directory in the grid is selected. Assuming both things are true, the full path to the item to be deleted is gotten using the `getFullPath()` method we saw earlier. Then, a quick confirmation pop-up is presented because nobody likes accidentally deleting things from the file system! Assuming the user accepts the deletion, a call to the `deleteFile()` method of the `FileSystemFunctions` class is called, passing that full path we got previously. The callback function then has two tasks to perform. First, the file grid needs to be updated because that definitely changed whether it was a file or directory. This is accomplished by the call to `directoryClicked()`, passing it the currently selected directory. Second, the directory tree needs to be updated because it may have been a directory that was deleted. (This is admittedly an inefficient design, because we should be able to tell whether it was in fact a directory or not and skip this step if it was a file, since the directory tree wouldn't change in that case; but this approach makes the code more concise, and let's face it, this chapter is already pretty long!) Before doing the actual update via the call to `directoryExpanded()`, we need to delete the currently selected directory node from the tree, add a "Loading" node in its place (mimicking how the tree was when first created) and then call the `directoryExpanded()` method to effectively regenerate that branch of the tree that represents the directory that may have changed.

doUpOneLevel() Method

The "up one level" function allows the user to jump to the parent of the currently selected directory with the click of a single button. The code that accomplishes that is as follows:

```
this.doUpOneLevel = function(inSelectedDirectory) {

  // Only do something if a directory is selected.
  if (inSelectedDirectory == null || inSelectedDirectory == "") {
    return;
  }

  // Get the parent of the currently selected directory node and select
  // it and refresh the grid, if the directory has a parent (file system
  // roots wouldn't).
```

```
  var parentNodeID = directoryTree.getParentId(inSelectedDirectory);
  if (parentNodeID != 0) {
    directoryTree.selectItem(parentNodeID);
    this.directoryClicked(parentNodeID);
  }

} // End doUpOneLevel().
```

First, a quick check to be sure a directory is currently selected. Then, we use the `getParentId()` method of the tree component to get the parent of the currently selected directory. If that method returns `0`, there is no parent (a file system root is selected), and we're done. Otherwise, we call the `selectItem()` method of the tree component, passing it the parent node ID we previously got. Finally, a call is made to `directoryClicked()`, which has the effect of populating the file grid with the contents of the parent directory, since it's now the currently selected directory.

doNewFile() Method

Creating a new file involves a call to the relatively simple `doNewFile()` method:

```
this.doNewFile = function(inSelectedDirectory) {

  // Only do something if a directory is selected.
  if (inSelectedDirectory == null || inSelectedDirectory == "") {
    alert("Please select a directory to create new file in");
    return;
  }

  // Get directory path where the new file will be created.
  var directoryPath = directoryTree.getUserData(inSelectedDirectory, "path");

  FileSystemFunctions.createFile(directoryPath,
    { callback : function(inResp) {
        fileman.directoryClicked(inSelectedDirectory);
      }
    }
  );

} // End doNewFile().
```

Once a check to be sure a directory is currently selected is passed, a call to the `createFile()` method of the `FileSystemFunctions` class is made. The current directory path (retrieved with the `getUserData()` method of the tree component) is passed to the `createFile()` method. Once the response comes back, the selected directory is refreshed via a call to `directoryClicked()` so that the new file appears there.

WHAT ABOUT EXCEPTIONS, YOU ASK?

You'll note that in none of these methods we're discussing are error conditions handled explicitly. Remember that we configured a global exception handler earlier, so there's no need to do anything more in any of these methods. Any exception thrown by the server will be handled generically, no questions asked.

It's my opinion that if you can design your applications such that a single global exception handler can do the trick for you, that's definitely a Good Thing™ since it will greatly simplify the code throughout the application. Of course, such a single generic function might not always be appropriate; but if you start out that way, you can still configure a handler for a specific call that needs to handle things a little differently, and it will override the global handler. You may even want to do your call-specific work, and then call the generic handler yourself. That would be a good, extensible design for sure.

doNewDirectory() Method

Creating a new directory is only slightly more involved than creating a new file, due to the need to update the directory tree as well as the file grid.

```
this.doNewDirectory = function(inSelectedDirectory) {

  // Only do something if a directory is selected.
  if (inSelectedDirectory == null || inSelectedDirectory == "") {
    alert("Please select a directory to create new directory in");
    return;
  }

  // Get directory path where the new directory will be created.
  var directoryPath = directoryTree.getUserData(inSelectedDirectory, "path");

  FileSystemFunctions.createDirectory(directoryPath,
    { callback : function(inResp) {
        // Refresh both the directory file list and the directory tree.
        fileman.directoryClicked(inSelectedDirectory);
        directoryTree.deleteChildItems(inSelectedDirectory);
        directoryTree.insertNewChild(inSelectedDirectory,
          inSelectedDirectory + "_dummy", "Loading...", 0, "blank.gif",
          0, 0, 0, 0);
        fileman.directoryExpanded(inSelectedDirectory);
      }
    }
  );

} // End doNewDirectory().
```

First, a quick check to be sure a directory is currently selected. Then, the current directory path (retrieved with the `getUserData()` method of the tree component) is retrieved, and is passed to the `createDirectory()` remote method of the `FileSystemFunctions` class. The

callback function then calls `directoryClicked()` to update the file grid. Then, the directory tree is updated, but as we've seen before, only after the currently selected directory node is deleted from the tree, and a "Loading" node is added. Finally, a call to the `directoryExpanded()` method is made, regenerating the branch of the tree that represents the directory that now has the new child to be displayed as part of the tree.

doEditFile() Method

Editing a file, the task performed by the `doEditFile()` method, amounts to not much more than getting the text of the file to be edited, showing the file editor `<div>`, and storing some information that we'll need when the user clicks the Save File button.

```
this.doEditFile = function(inSelectedDirectory, inSelectedItem) {

  // Only do something if both a directory and an item in the grid are
  // selected.
  if (inSelectedDirectory == null || inSelectedItem == null ||
    inSelectedDirectory == "" || inSelectedItem == "") {
    alert("Please select a file to edit");
    return;
  }

  // Stop editing of directories.
  var itemType = fileGrid.cells(inSelectedItem, 2);
  if (itemType.getValue() == "Directory") {
    alert("Sorry, but only files can be edited");
    return;
  }

  // Show the file editor display and set the textarea text to "Loading...".
  dwr.util.byId("divFileEditor").style.display = "block";
  dwr.util.setValue("taFileEditor", "Loading...");

  // Get fully qualified name of selected item.
  var fullPath = fileman.getFullPath(inSelectedDirectory, inSelectedItem);

  fileBeingEdited = fullPath;
  FileSystemFunctions.editFile(fullPath,
    { callback : function(inResp) {
        dwr.util.setValue("taFileEditor", inResp);
      }
    }
  );

} // End doEdit().
```

As usual, we first verify that a directory and a file are in fact selected; otherwise, there's nothing to do here, and we immediately exit the function. Next, we get the contents of the

third column in the file grid, which is the type of the selected item. If it turns out that it's a directory, we pop up a message saying directories can't be edited, and the function is exited.

The next task is to show the file editor `<div>`, and to do this we use the `dwr.util.byId()` method to get a reference to it, and then change its display style attribute to `block`. In addition, the text "Loading..." is placed in the `<textarea>` so the user has something to look at while the text of the file is retrieved.

Then, we use the `getFullPath()` method again to get the full path of the file to be edited. Lastly, with that full path in hand (and stored in the `fileBeingEdited` field of the `Fileman` class), we call the `editFile()` method of the `FileSystemFunctions` class. This returns to us the text of the file as a string, which we then set in the `<textarea>` using the `dwr.util.setValue()` function. At this point, the screen looks like what you see in Figure 6-13.

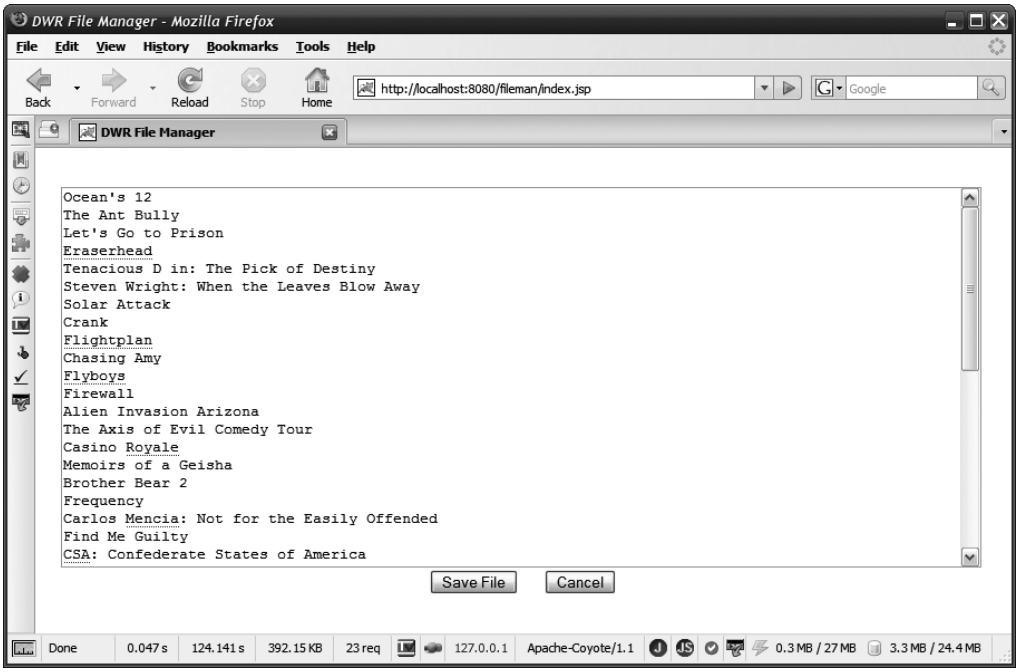


Figure 6-13. *The view as seen when editing a file, namely the contents of my Netflix[5] queue. (Some of them are my wife's picks, so you can't blame me for all of them!)*

The user can then cancel editing or click the Save File button to save changes, which invokes the `doSaveFile()` method.

5. Netflix (`www.netflix.com`) is a web site that offers movie rentals for a monthly fee. You can have DVDs shipped to your home, or recently, you can even download movies to watch instantly. The price is quite good on most of their various plans, and the convenience of never having to go to a video store again is well worth it.

doSaveFile() Method

Saving the file once the user clicks the Save File button is a trivial matter, as you can see in the `saveFile()` method here:

```
this.saveFile = function() {

  FileSystemFunctions.saveFile(fileBeingEdited,
    dwr.util.getValue("taFileEditor"),
    { callback : function(inResp) {
        dwr.util.byId("divFileEditor").style.display = "none";
        alert("File has been saved");
      }
    }
  );

} // End saveFile().
```

It's simply a call to the `saveFile()` method of the `FileSystemFunctions` class, passing it the fully qualified file being edited, and the text in the `taFileEditor` `<textarea>`, and that's it. When the response returns, the file editor `<div>` is hidden, and the user is alerted that the changes were saved. Couldn't be easier!

doDownload() Method

As you previously saw, downloading a file is implemented in the `downloadFile.jsp` file, so the question is, how do we get the user there? The answer is found in the `doDownload()` method:

```
this.doDownload = function(inSelectedDirectory, inSelectedItem) {

  // Only do something if both a directory and an item in the grid are
  // selected.
  if (inSelectedDirectory == null || inSelectedItem == null ||
    inSelectedDirectory == "" || inSelectedItem == "") {
    alert("Please select a file to download");
    return;
  }

  // Make sure the selected item is a file.
  var itemType = fileGrid.cells(inSelectedItem, 2).getValue();
  if (itemType == "Directory") {
    alert("Sorry, only files can be downloaded");
    return;
  }

  // Get the name and path of the selected file.
  var directoryPath = directoryTree.getUserData(inSelectedDirectory, "path");
  var fileName = fileGrid.getUserData(inSelectedItem, "name");
```

```
  // Redirect to the file.  This won't cause the current page to be lost,
  // because the content disposition set by the server is set to download.
  window.location = "downloadFile.jsp?" +
    "p=" + encodeURIComponent(directoryPath) +
    "&n=" + encodeURIComponent(fileName);

} // End doDownload().
```

As usual, we first verify that a directory and a file are in fact selected; otherwise, there's nothing to do here, and we immediately exit the function.

Next, we get the contents of the third column in the file grid, which is the type of the selected item. If it turns out that it's a directory, we pop up a message saying directories can't be downloaded, and the function is exited.

After that, we get the path of the selected directory from the tree component. Then we get the name of the selected file from the file grid. With that information, we can now get to the `downloadFile.jsp`. To do this, we set the location of the `window` object to a URL we construct that includes two parameters: `p`, which is the path, and `n`, which is the name of the file. You may be wondering why the `Fileman` app itself doesn't get overwritten when we do this, and the answer is that the `downloadFile.jsp`, as you may recall, sets the `Content-Disposition` header to indicate the response is to be downloaded, so the browser won't overwrite what's there already, and Fileman lives to, err, file, another day!

doUpload() Method

The `doUpload()` method is next, and it's a very quick and easy hit:

```
this.doUpload = function(inSelectedDirectory) {

  // Only do something if a directory is selected.
  if (inSelectedDirectory == null || inSelectedDirectory == "") {
    alert("Please select a directory to upload to");
    return;
  }

  // Get the path of the directory and set it in the form.
  var directoryPath = directoryTree.getUserData(inSelectedDirectory, "path");
  dwr.util.setValue("uploadDirectory", directoryPath);

  // Open the file upload form.
  dwr.util.byId("divFileUpload").style.display = "block";

} // End doUpload().
```

All this method needs to do is (a) ensure that a directory is selected, so that we can tell the server where to place the file, (b) get the path to that directory and place it in the file upload form in the field `uploadDirectory`, and (c) show the form. The user can then select the file to upload, or cancel. Figure 6-14 shows the upload form as the user sees it.

DWR File Manager - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Back Forward Reload Stop Home
http://localhost:8080/fileman/index.jsp
Google
DWR File Manager
Upload a file
File to upload:
Browse...
Upload file
Cancel
D... 0.047 s 25.610 s 390.60 KB 21 req 127.0.0.1 Apache-Coyote/1.1 0.3 MB / 27 MB 3.3 MB / 24.4 MB

Figure 6-14. *The file upload form*

Simplistic, yes, but it accomplishes its goal just fine (although be sure to look at the "Suggested Exercises" section for a good, fairly obvious suggestion around this form).

doAbout() Method

To display the About dialog box, the `doAbout()` method is invoked:

```
this.doAbout = function() {

  // Just a simple alert pop-up.
  alert("DWR File Manager v1.0\nAugust 11, 2007\n\nBy Frank W. Zammetti\n\n" +
    "As it appeared in the book " +
    "\"Practical DWR 2 Projects\"\n" +
    "Published by Apress, Inc. (ISBN 1-59059-941-1)");

} // End doAbout().
```

As you can see, it's nothing but an `alert()` function call. Not the stuff of senior thesis papers, I'll tell you that!

doPrintDirectoryContents() Method

The `doPrintDirectoryContents()` method implements the one tool found on the Tools menu, which is effectively a printable version of the current contents of the file grid:

```
this.doPrintDirectoryContents = function(inSelectedDirectory) {

  // Only do something if a directory is selected.
  if (inSelectedDirectory == null || inSelectedDirectory == "") {
    alert("Please select a directory to print the contents of");
```

```
    return;
  }

  // Get path of clicked directory.
  var path = directoryTree.getUserData(inSelectedDirectory, "path");

  // Clear grid and put in loading message.
  FileSystemFunctions.listFiles(path,
    { callback : function(inResp) {
        // Iterate over the collection of FileVO objects returned.
        var htm = "<html><head><title></title></head>" +
          "<body><table border=\"0\" width=\"100%\" " +
          "style=\"font-size:8pt;font-weight:bold;\"><tr>";
        for (var i = 0; i < inResp.length; i++) {
          // For each, generate markup to display and then insert it.
          htm += "<tr valign=\"top\"><td>" + inResp[i].name + "</td><td>" +
            inResp[i].size + "</td><td>" + inResp[i].type + "</td><td>" +
            inResp[i].modified + "</td></tr>";
        }
        htm += "</table></body></html>";
        var printDialog = window.open("", "", "width=780,height=500");
        printDialog.document.open()
        printDialog.document.write(htm)
        printDialog.document.close()
        printDialog.print();
        printDialog.close();
      }
    }
  );

} // End doPrintDirectoryContents().
```

Once we know there is actually a selected directory, we get the path of that directory, and use the `listFiles()` method of the `FileSystemFunctions` class, just like in the `directoryClicked()` method previously. This time though, instead of populating the grid with the returned collection of `FileVO` objects, we'll instead construct a simple HTML document out of it with a table in place of the grid. Once that HTML is fully constructed, we open a pop-up window and write the HTML out to it, and then call the `print()` method of that window, and we have ourselves a tool to print the contents of the current directory, just like that!

The Server-Side Code

With the client-side code out of the way, we can now begin to look at the server-side code, and as you'll see, there's not a whole lot of it to look at!

FileVO.java

The `FileVO` class is a value object that describes a single file. Rather than use the Java standard `File` class, I created my own VO so that I could include only that information I was interested in about a file, and any extra that the `File` class didn't provide.

The `FileVO` class is basically just a typical javabean with private fields and a bunch of getters and setters. You'll also find my usual `toString()` implementation floating around. The one exception to this typicality (is that a word?) is the `setType()` method, which is this code:

```
public void setType(final String inType) {

  // If inType is null, we'll dynamically try and detect the file type.
  // Otherwise we'll just set the type to the value of inType.
  if (inType == null && name != null) {
    String fileExtension = "";
    int dotLocation = name.lastIndexOf(".");
    if (dotLocation != -1) {
      fileExtension = name.substring(dotLocation + 1);
    }
    type = "Unknown File";
    if (fileExtension.equalsIgnoreCase("txt")) {
      type = "Text Document";
    } else if (fileExtension.equalsIgnoreCase("zip")) {
      type = "Zip Archive";
    } else if (fileExtension.equalsIgnoreCase("pdf")) {
      type = "Adobe Acrobat Document";
    ...
    } else if (fileExtension.equalsIgnoreCase("sh")) {
      type = "Shell Script";
    } else if (fileExtension.equalsIgnoreCase("csv")) {
      type = "Comma-Separated Values File";
    }
  } else {
    type = inType;
  }

} // End setType().
```

I cut out a whole bunch of `else if` branches in between, noted by the `...` line. What happens here is that when a `FileVO` object is instantiated and populated, the type is either sent in or isn't, depending on what code is doing the instantiating and populating. If `null` is passed in, the `setType()` will attempt to detect the file type based on extension. This will generally work pretty well in Windows, where file extensions are typical, but may not work in other operating systems where they are less common. In any case, we locate the file extension portion of the file name, if there is any (which means that `setType()` must be called after `setName()`, or this won't work), and then branch on it for a number of relatively common file extensions. We default to the value "Unknown File", and that's what the user will see if the file type isn't recognized, or if the file didn't have an extension.

Figure 6-15 shows the UML class diagram for the `FileVO` class. Like I said, it's a perfectly ordinary javabean, save for the `setType()` method. The four fields you see should be self-explanatory, as they match the columns the user sees on the screen.

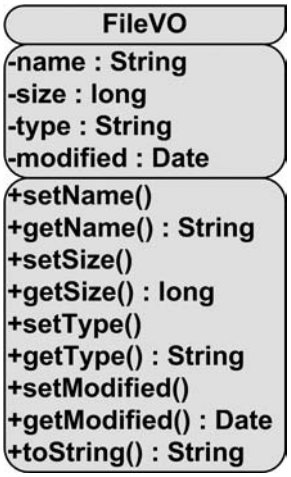


Figure 6-15. *UML class diagram for the* `FileVO` *class*

DirectoryVO.java

The `DirectoryVO` class, just like the `FileVO` class, is an ordinary javabean that provides some extra information the `File` class wouldn't, and gets rid of all the superfluous information. Figure 6-16 shows the UML class diagram for this class.

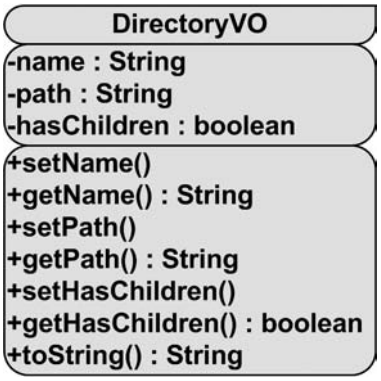


Figure 6-16. *UML class diagram for the* `DirectoryVO` *class*

It's obvious, I think, what the `name` and `path` fields are. The `hasChildren` field is set to `true` when the directory has child directories, `false` when it doesn't. As we saw earlier, this is used to set up the UI properly in terms of branches being expandable or not in the directory tree.

To save some space, the code for the `DirectoryVO` class isn't shown here, but have a look anyway. It literally is just the private members, getters and setters, and `toString()`. There's not even an interesting setter like in `FileVO`!

FileSystemFunctions.java

We saw as we looked over the client-side code that all the calls to the server via DWR were to the `FileSystemFunctions` class, and now it's time to check that class out and see how all the features of Fileman are actually implemented.

To begin, have a look at Figure 6-17, which is the UML class diagram of the `FileSystemFunctions` class. It gives you an overview of what's present in the class.

Figure 6-17. *UML class diagram for the* `FileSystemFunctions` *class*

As you can see, the methods present are those we saw calls to in the client-side code, and there's not really a whole lot of them in total. One thing that the diagram doesn't show is that this class extends `org.apache.commons.io.DirectoryWalker`, which was introduced in the "Jakarta Commons IO" section. The reason for this extension is so that the `handleDirectory()` method is present and the class can be used to walk through a directory structure and process each element encountered, and we'll begin by looking at that very method.

handleDirectory() Method

The `handleDirectory()` method is an event callback that will be called for every item encountered as a directory tree is walked. The code of that method is this:

```
@SuppressWarnings("unchecked") // Avoids warning on inResults.add(dv)
protected boolean handleDirectory(final File inDirectory, final int inDepth,
  final Collection inResults) {

  // Only process directories that are direct children of inDirectory.
  if (inDepth <= 1) {
    // Don't process root file system directories.
    if (inDirectory.getParent() != null) {
```

```
      // Don't process the directory if it's the requested directory.
      if (!directory.equals(inDirectory.getPath())) {
        // Construct DirectoryVO object.
        DirectoryVO dv = new DirectoryVO();
        dv.setName(inDirectory.getName());
        dv.setPath(inDirectory.getPath());
        // See if the directory has child directories or not.  Anonymous inner
        // class used to determine whether the directory has children or not.
        String[] childDirectories = inDirectory.list(
          new FilenameFilter() {
            public boolean accept(final File inDir, final String inName) {
              if (inName != inDirectory.getName()) {
                File f = new File(inDir.getPath() + File.separator + inName);
                if (f.isDirectory()) {
                  return true;
                } else {
                  return false;
                }
              } else {
                return false;
              }
            }
          }
        );
        if (childDirectories != null && childDirectories.length > 0) {
          dv.setHasChildren(true);
        }
        inResults.add(dv);
      }
    }
    return true;
  } else {
    return false;
  }

} // End handleDirectory().
```

To begin with, take note of the annotation used on the method. This is present to avoid a compile warning due to the fact that the `inResults` parameter is not strongly typed using generics. The code works just fine with or without the annotation; it just gets rid of a (in this case) superfluous warning.

Inside the method, the first thing you encounter is a check of the `inDepth` parameter. As a directory is walked, all its children, and all their children, and all their children, and so on, will trigger a call to this method. However, for our purposes here, we only care about directories that are direct children of a given directory, and in that case, `inDepth` is either `0` or `1` (`0` accounts for the directory itself . . . it might seem that `== 1` would be the right check here, but in fact that doesn't work because it effectively ignores the directory we started with, essentially aborting the walking for all practical purposes).

Once we know we should process this directory, we then need to abort if the directory is a root file system or the directory we started with. In neither case would we want to add that to the list of subdirectories returned to the UI.

Once these checks are passed, it's time to go ahead and process the directory. Doing so involves instantiating a `DirectoryVO` object and populating its `name` and `path` attributes. Next, we need to determine whether the directory itself has child directories so we can set the `hasChildren` field appropriately. To do that, we use the `list()` method of the `File` object representing the current directory being processed, which is passed in as the `inDirectory` parameter. To that `list()` method we pass an anonymous class implementing the `FilenameFilter` interface. The problem is that the `list()` method includes returning the directory we're checking itself, so we'd always get a false positive if we allowed that to be checked since obviously we know for sure that's a directory. So, the filter implementation rejects that match, and for all others it then uses the `isDirectory()` method of the `File` class, and if it returns `true`, that directory is added to the array being built up by the `list()` method. Finally, we check the length of that array, and if it's `1` or more, `hasChildren` is set to `true`, just like we want.

listFiles() Method

The `listFiles()` method is one of two methods used to get a list of items for the UI to display; this one returns both files and directories under a given directory.

```
public List listFiles(final String inDirectory) throws Exception {

  try {
    File[] files = new File(inDirectory).listFiles();
    List<FileVO> results = new ArrayList<FileVO>();
    if (files != null) {
      for (File f : files) {
        // Construct a FileVO for each file (or directory).
        FileVO fv = new FileVO();
        fv.setName(f.getName());
        fv.setSize(f.length());
        if (f.isDirectory()) {
          fv.setType("Directory");
        } else {
          fv.setType(null);
        }
        fv.setModified(new Date(f.lastModified()));
        results.add(fv);
      }
    }
    return results;
  } catch (Exception e) {
    throw new Exception("Exception occurred: " + e);
  }

} // End listFiles().
```

The path to get the list for is passed in as `inDirectory`, so the code begins by creating a `File` object from that and calling the `listFiles()` method on it to get a raw list of files and directories. Then, that list is iterated over, and for each a `FileVO` object is constructed that includes the name, size, last modified date/time, and whether the item is a file or directory. This object is added to a `List`, and that `List` is returned by the method.

listDirectories() Method

The `listDirectories()` method complements the `listFiles()` method but returns a list strictly of directories. This is also the method that triggers the use of the `handleDirectory()` event callback method shown previously.

```
public List listDirectories(final String inStartDirectory) throws Exception {

  directory = inStartDirectory;
  List<DirectoryVO> results = new ArrayList<DirectoryVO>();
  try {
    // Walk the directory tree, starting at inStartDirectory.  The results
    // list will be populated as the walking is done.
    walk(new File(inStartDirectory), results);
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  return results;

} // End listDirectories().
```

First, the directory to list subdirectories for is stored in the class-level `directories` field. This is necessary because the `handleDirectory()` method will need this information, but we have no other way to pass it directly to it. (A `ThreadLocal` might have been another answer, perhaps even a better one, but it also complicates the code a little more than I'd like.) Next, a new `List` is instantiated, which will be what gets returned ultimately from this method. Then, the `walk()` method is called, which is inherited from the `DirectoryWalker` class. Its execution triggers the calls to `handleDirectory()` that we've discussed before. Finally, the `List` is returned, and our job here is done!

listRoots() Method

The `listRoots()` method is called from the client during execution of the `init()` method of the `Fileman` JavaScript class:

```
public List listRoots() throws Exception {

  try {
    File[] roots = File.listRoots();
    List<DirectoryVO> results = new ArrayList<DirectoryVO>();
    for (final File f : roots) {
      DirectoryVO dv = new DirectoryVO();
      dv.setName(f.getPath());
```

```
      dv.setPath(f.getPath());
      // See if the directory has child directories or not.  Anonymous inner
      // class used to determine whether the directory has children or not.
      String[] childDirectories = f.list(
        new FilenameFilter() {
          public boolean accept(final File inDir, final String inName) {
            if (inName != f.getName()) {
              File f = new File(inDir.getPath() + File.separator + inName);
              if (f.isDirectory()) {
                return true;
              } else {
                return false;
              }
            } else {
              return false;
            }
          }
        }
      );
      // Setting whether the directory has children or not is used by the UI
      // code to set up the grid properly in terms of nodes being expandable
      // or not.
      if (childDirectories != null && childDirectories.length > 0) {
        dv.setHasChildren(true);
      }
      results.add(dv);
    }
    return results;
  } catch (Exception e) {
    throw new Exception("Exception occurred: " + e);
  }

} // End listRoots().
```

The `listRoots()` method of the basic Java `File` class provides everything we need. From that point on, this method works very much like the `listDirectories()` method we already examined, so I'll skip repeating myself too much.

createDirectory() Method

The `createDirectory()` method is used to . . . wait for it . . . create a new directory!

```
public void createDirectory(final String inPath) throws Exception {

  boolean outcome = false;
  try {
    outcome =
      new File(inPath + File.separator + "new_directory").mkdir();
  } catch (Exception e) {
```

```
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  if (!outcome) {
    throw new Exception("Directory could not be created\n\n" +
      "(Does a directory with the name 'new_directory' already exist?");
  }

} // End createDirectory().
```

As you can see, it's just using the standard Java `File` class's `mkDir()` method to create the directory. Note that only the path to the parent directory is passed in. The new directory is given the generic name `new_directory` (very creative of me, huh?). The user can, of course, rename this immediately, and in all probability will do exactly that, but then it's just a simple rename operation, so no biggie. An exception is thrown if anything goes wrong, and since the most likely cause of the exception is that a directory with that name already exists, the error message provides that as a hint.

deleteFile() Method

Like creating a file, deleting a file involves nothing but standard Java file I/O code:

```
public void deleteFile(final String inFullPath) throws Exception {

  boolean outcome = false;
  try {
    outcome = new File(inFullPath).delete();
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  if (!outcome) {
    throw new Exception("File/directory could not be deleted\n\n" +
      "Possible permission issue?\n\n" +
      "(If it's a directory, it must be empty to be deleted)");
  }

} // End deleteFile().
```

Once again, the ubiquitous `File` class provides a ready-made `delete()` method for us to use. As opposed to `createFile()`, we get the fully qualified path of the file (or directory) to delete. Once again, we throw an exception on any problem, and we give a hint about the directory not being empty, since that is the most likely cause of a failure (the account the app server is running under not having delete permissions or the directory not being empty are probably the two most likely causes, so both are mentioned).

renameFile() Method

The `renameFile()` method is another quick hit:

```
public void renameFile(final String inOldFullPath,
  final String inNewFullPath) throws Exception {

  boolean outcome = false;
  try {
    outcome = new File(inOldFullPath).renameTo(new File(inNewFullPath));
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  if (!outcome) {
    throw new Exception("File/directory could not be renamed\n\n" +
      "(Does a file/directory with the new name already exist?");
  }

} // End renameFile().
```

This time, we get the full path to both the file that is being renamed and the file as it will appear after the rename. We then simply use the `renameTo()` method of the `File` class to do the actual work. The most likely exception cause is a file with that name already being present in the directory, so that's the hint we give in the error message.

createFile() Method

Here we have another in the long parade of very simple functions; this time it's creating a file via the `createFile()` method:

```
public void createFile(final String inPath) throws Exception {

  try {
    File f = new File(inPath + File.separator + "new_file.txt");
    if (!f.createNewFile()) {
      throw new Exception("File already exists");
    }
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }

} // End createFile().
```

This is, for all intents and purposes, the same as creating a new directory, so I'm only going to mention one difference, that being the use of the `FileUtils` Commons IO class, specifically its `writeStringToFile()` method. Just instantiating the `File` instance doesn't create the file; we have to actually write something to it. Of course, we don't have any real content yet, so an empty string suffices nicely.

WAIT, WE HAVE COMMONS IO, DON'T WE?

You might be wondering why I didn't use Commons IO classes for the last four methods described. All of them could have been done with Commons, but I didn't see the point when the basic `File` class provides exactly what we need. In fact, some of these methods would have been a few lines longer had I used Commons code. If you're interested in playing with Commons a bit more, this would be a great exercise to undertake. This goes not only for the last four methods described, but most of the others as well.

copyMoveFile() Method

The `copyMoveFile()` method is, of course, what gets called when the user initiates the paste operation from the UI:

```
public void copyMoveFile(final String inSourcePath,
  final String inSourceName, final String inDestinationPath,
  final String inOperation, final String inType) throws Exception {

  boolean outcome = true;
  try {
    File src = new File(inSourcePath + File.separator + inSourceName);
    File dest = new File(inDestinationPath + File.separator +
      (inOperation.equals("copy")?"copy_of_":"") + inSourceName);
    // Different copy method based on whether copying file or directory.
    if (inType.equals("file")) {
      FileUtils.copyFile(src, dest);
    } else {
      FileUtils.copyDirectory(src, dest);
    }
    // Delete the original file if doing a cut.
    if (inOperation.equals("cut")) {
      deleteFile(inSourcePath + File.separator + inSourceName);
    }
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  if (!outcome) {
    throw new Exception("File/directory could not be copied/moved\n\n" +
      "(Does the file/directory already exist in the destination?)");
  }

} // End copyMoveFile().
```

We begin by creating a `File` object representing the source file (or directory) and one representing the destination directory. Note the trinary logic line when creating the destination: when we're pasting a file, we need to give it a name, and in this case it's `copy_of_xxxx` where

xxxx is the original file name. Then, a simple branch based on whether we're dealing with a file or a directory determines which method of the Commons IO `FileUtils` class will be used: `copyFile()` for files, `copyDirectory()` for directories. Once that's done, for cut operations, we need to delete the original file, which gives us the illusion of a cut, so we use the `deleteFile()` method we already saw to take care of it.

editFile() Method

Editing a file is another one of those very, very simple methods that we can get through in a heartbeat:

```
public String editFile(final String inPath) throws Exception {

  String fileContents = null;
  try {
    fileContents = FileUtils.readFileToString(new File(inPath));
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }
  return fileContents;

} // End editFile().
```

The Commons IO `FileUtils` class contains a nice `readFileToString()` method that, as its name indicates, results in a string read in from a file. Perfect for our purposes! So, read in the file, get a string, and return it from the method, and we're good to go!

saveFile() Method

When the user wants to save edits to a file, the `saveFile()` method is ultimately called upon to do that work.

```
public void saveFile(final String inPath, final String inText)
  throws Exception {

  try {
    File f = new File(inPath);
    FileUtils.writeStringToFile(f, inText);
  } catch (Exception e) {
    e.printStackTrace();
    throw new Exception("Exception occurred: " + e);
  }

} // End saveFile().
```

We already saw the `FileUtils.writeStringToFile()` method in the `createFile()` method before, so this method should be entirely self-explanatory.

Whew, we made it! Another project in the books, quite literally! This touched on a number of good topics and gave us a look at a few more facets of DWR, and gave us a handy little utility application that you could use in real life (as opposed to fake life, I suppose?).

But wait, there's always room for improvement, so . . .

Suggested Exercises

It's definitely not too hard to come up with suggestions for this project. All you really need to do is try out any of the file managers mentioned in the "A Bevy of File Managers" sidebar early in this chapter and start pulling out features! A good file manager is worth its weight in gold, and it is my hope that there is a good foundation here to build upon. So, let's see what suggestions I can come up with to not only make the application better, but get you more experience with DWR.

- Create a function to compare two files. Commons IO includes such a function, so it shouldn't be very hard to implement on the server side. On the client, you'll need to dig into the API documentation for the dhtmlxGrid component and figure out how to allow multiple selections (hint: it's easy!). Turn on that ability temporarily so the user can select two files, do the compare, and then turn it off.
- Even better, based on the last suggestion, how about give the user the ability to **always** select multiple files or directories for operations where it makes sense? Windows Explorer obviously allows this, and I left it out to make the code simpler, but also to leave it as an exercise for you. This will be a bit of a challenge because you'll need to alter the method signatures for various server-side functions to accept an array of selections, but otherwise I wouldn't expect it to be too difficult.
- Build a tool to get the size of a selected directory. This is a really handy thing in a file manager. Bonus points if you add a column to the grid and allow the ability to show the sizes of all child directories of the current directory being viewed!
- How about the ability to view and alter file and directory attributes? It could be a little challenging if you intend to keep the app cross-platform.
- Replace the sparse file upload form with the dhtmlxFileVault component. Frankly, I only thought of doing this after I had already completed the project; otherwise, I likely would have done it myself!
- Add a Favorites menu that dynamically grows as items are added to it, so the user can quickly jump to common locations. It would also be nice to allow the user to save that list on the server for future sessions.

Summary

So, if you decide to put a server up at home, assuming your ISP allows it, you now have yourself a nifty little file manager to run on it! In the process of building this application, we saw more of DWR in action, including a new twist on securing remotable classes, namely the "deny by default" paradigm. We saw how a good set of UI components can make creating a robust,

visually pleasing interface a breeze. We saw how combining those two things, plus a few extra libraries on the back end, can allow you to create powerful, rich Internet-based applications that in many ways rival their fat-client equivalents.

In the next chapter, we'll develop a project that is a little more "Enterprise" in nature, something you can imagine seeing in any office in America (or any other country for that matter), namely a reporting portal. With DWR being used, plus the addition of a nice open source reporting package, you'll wind up with an application you could throw at your boss and get yourself that raise you almost certainly deserve!

CHAPTER 7

Enter the Enterprise: A DWR-Based Report Portal

If you're a true geek like me, the title of this chapter got you excited thinking about the NCC-1701D and all the coolness it implies. If you're a true *Star Trek* fan, you've read the technical manual for the *Enterprise* and have most of it committed to memory . . . if not, that's one nerdy demerit for you, and you should say ten Hail Picards in penance!

Alas, that's not what this chapter is about, although I hope you'll find this chapter pretty cool anyway. Instead of creating our ticket to the stars (not to mention the holodeck, the invention of which will without question result in the fall of mankind long before we reach the stars!), we'll be building an application that you'll often find in another type of enterprise (the kind concerned with making money for shareholders more than the exploration of the final frontier), the application being a report portal.

Report portals are all the rage: they give users a single-stop location to run reports, schedule them, and see the results of previous report runs. Using DWR, and a few other libraries as well that we'll explore as we go, we can put together a rather nifty little portal with most of the basic functions such a portal typically has, and do so without too much difficulty. In this chapter, you'll see some new DWR tricks, including integration with Spring and some more capabilities in the `dwr.util` package, and you'll also get an introduction to a great open source reporting package named DataVision. You'll also get some exposure to script.aculo.us, a very popular client-side library for effects, and we'll dig into Spring a bit further as well.

So, I'll now use what should be an all-too-obvious *Star Trek* cliché to get us going: "Warp 9, ENGAGE!"[1]

Application Requirements and Goals

So, what precisely is a report portal anyway? Simply put, it's a web site, roughly in the form of a portal (i.e., a page with subsections that can generally be manipulated in terms of turning them on and off and which function almost like individual applications) that allows the user

1. If you like *Star Trek*, you'll likely love this chapter as it's my way of paying homage to something that I've loved my whole life . . . of course, if you're not a Trekker or Trekkie, whichever term you prefer, you're probably not a carbon-based life form from the planet Earth anyway. I do realize that not everyone is a fan though, so I'll try and explain all the obscure references as best I can. That's not to say there won't be some inside jokes that only fans will likely get, but hey, how can you be a programmer if you're not a *Star Trek* fan anyway?!?

to view various reports, schedule reports, and maintain reports. If you've ever used a product like Crystal Info or Business Objects, you'll know what I'm talking about. If you haven't, the preceding description should give you the rough picture. Basically, it's a dashboard-type site, most usually in an enterprise environment, where business reports are made available to users as required.

So what exactly is our version, which we'll call RePortal, going to do? Here's the rundown of our goals:

- First, obviously, users should be able to create reports, run reports on demand, schedule reports, and view the output of scheduled runs. To do all this, we'll use an open source reporting tool called DataVision for our reporting needs. We'll get to that product shortly. When talking about running reports on demand, let's all give users a favorites list so they can have quick access to the reports they care about. Let's further ensure that we have a capability to have some reports available to all users, and some reports available only to certain users.

- To meet the preceding requirement, we'll need at least some rudimentary security mechanism. We'll be able to create and delete users, as well as user groups, and assign users to groups. We'll base what reports a given user sees on what group that user belongs to.

- We'll also need to deal with the scheduling aspect of things. We could simply spawn a thread to handle it, but that doesn't seem quite "enterprise" enough, does it? Let's go with something a little fancier: an "enterprise" scheduler by the name of Quartz.

- We'll break the UI into various subsections, since it's a portal we're building (it's not the atavachron,[2] but it'll do). So, each of these sections should be closable, and also should be able to expand and contract to make more room on the screen as the user desires.

- With all this expanding and contracting, let's be Web 2.0 a bit and throw in some effects. To do so, we'll use the very popular script.aculo.us library. Let's also use what's commonly referred to as a *lightbox* for a pop-up when the user logs in (if you've never heard that term, don't get your Klingon dreads in a bunch, it'll be explained).

- It seems obvious we'll need a database or two here, so let's use our old friend Derby once more, since it's served us so well in past projects.

- There's an opportunity here for seeing some DWR integration with other libraries in the form of using Spring's Dependency Injection capabilities, so let's play with that a bit.

With that set of goals and requirements in mind, let's now start taking a look at the various components we'll be rolling into this batter.

---

2. The atavachron was the time portal that appeared in the original *Star Trek* series' episode "All Our Yesterdays." This is, of course, only one of many time (and/or space) portals discovered in various *Star Trek* series. One would think any civilization with a grasp of fire could build one! "City on the Edge of Forever" is another famous episode that features such a portal, perhaps **the** most remembered one of all actually.

Spring Dependency Injection (IoC)

No, Dependency Injection, or DI for short, is not what happens when you marry someone that already has kids . . . nor is it anything you should be afraid to speak of in mixed company. It's a development pattern that basically says that instead of leaving a given class to instantiate all the other classes it depends on, let some container "inject" these dependencies into the class at the point of instantiation. The idea is to write less code, of course, and also to reduce coupling between classes (that part is only true to some extent, in my opinion, since the class still has to know about its dependencies, but it has to know **less**, which is good). DI is also referred to as Inversion of Control, or IoC, which is actually the name of the pattern being implemented. Both terms can generally be used interchangeably.

Spring was, if not the first, then certainly the implementation of this pattern that raised it to prominence. Spring has evolved over time to be a much richer platform encompassing much more than just IoC, but that's what it started out as, and what it really became popular as.

This is going to be a long chapter, so it won't be possible to go into great detail on DI, but I want to give you at least a brief overview of it.

DI comes in three common forms: setter injection, constructor injection, and interface injection. The setter injection form looks like this:

```
public class MyClass {
  private MyBean bean;
  public void setMyBean(MyBean inBean) {
    this.bean = inBean;
  }
}
```

When an instance of `MyClass` is instantiated via the IoC container, the container will create an instance of `MyBean` and call `setMyBean()` on the `MyClass` instance before any other methods are called. Very nice—all the other methods can then go ahead and use `MyBean` without having to have constructed the instance itself. Consider if `MyBean` required some complicated construction, for example, a bunch of fields set a certain way and things of that nature. Isn't it nice to not have had that code in `MyClass`? This type of setup of an injected object will generally be done via some form of configuration file, or annotations, so it's abstracted out from the code that is making use of it.

Another form of DI is constructor injection:

```
public class MyClass {
  private MyBean bean;
  public MyClass(MyBean inBean) {
    this.bean = inBean;
  }
}
```

It's not much different from setter injection, so it pretty much comes down to a matter of which code you prefer.

The final form is interface injection, which is only marginally different from the other two, but which requires two source files like so:

```
public interface MyInterface {
  public void setMyBean(MyBean inBean);
}
public class MyClass implements MyInterface {
  private MyBean bean;
  public MyClass(MyBean inBean) {
    this.bean = inBean;
  }
}
```

This is clearly a little different, and you may wonder why you'd want the extra code, and the answer is the same answer to why use interfaces in the first place. If you have a class that you want to have a known public API, and especially if you're going to have multiple implementations of that class that must expose the same API and therefore can be treated as one type (the type of the interface), then this is probably a desirable approach to DI.

At the end of the day though, this is all about getting dependencies into a class instance without the class having to do the work itself, and Spring makes this very easy. We'll see the details of doing this in Spring when we get to the application code, but you'll find it amounts to nothing more than an XML configuration file and a new option in `dwr.xml`.

As far as configuration of Spring goes, there's a couple of options including XML-based configuration and annotation-based configuration. This project will be using the XML-based configuration, and a Spring configuration file, commonly named `spring-beans.xml`, looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="groupWorker"
    class="com.apress.dwrprojects.reportal.GroupWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>

</beans>
```

Note that in Spring 2.0, this file is called `applicationContext.xml`.

We'll get into the particulars of what this all means as we explore RePortal's code, but I'd be willing to bet 300 quatloos[3] that you can figure most of it out just by staring at it for a moment.

As we look at how Spring is used in this application, you'll find that we're barely scratching the surface of what Spring has to offer in this area. To get more in-depth coverage of it, jump over to this link to the Spring user manual: `http://static.springframework.org/spring/docs/2.0.x/reference/beans.html`.

DataVision

There are plenty of reporting tools out there, and why not? Reporting is a huge part of what is done in IT departments at most businesses. I mean, what good is all that sales data, all that financial transaction information, if you can't look for trends and slice that data in various ways to get different perspectives on it? That's where reporting tools come in.

There are plenty of choices, many commercial offerings like Crystal Info and Business Objects, and a wide variety of open source packages like JasperReports and Pentaho Reporting (formerly JFreeReport). One of the oldest (roughly eight years old now) is DataVision.

DataVision (`http://datavision.sourceforge.net`) is a Java-based open source (Apache Software License 2.0) package that includes a Swing-based report designer, output in various formats (PDF, HTML, and Excel among them), the ability to easily integrate into both fat-client and web-based applications (as we'll see), and an ease of use that allows first-time users to get started quickly and easily, but with a great deal of flexibility and power as you get more advanced.

In the interest of full disclosure, I need to point out that last year I became the project lead of DataVision when the original developer, Jim Menard, decided to put his time into other things. To be fair though, I was using DataVision for at least two years prior to that and liked it a great deal before I ever got involved.

For our needs in RePortal, we'll just really be using the ability of DataVision to be integrated into our application. However, we'll also need a sample report to play with, and that is included in the downloadable code for this book. DataVision stores its reports in a straightforward XML format, so you can always edit them by hand, but using the designer is definitely a better idea. Actually, using the designer is a topic for a whole other book (hey, Mr. Editor, are

---

3. In the original *Star Trek* series episode "The Gamesters of Triskelion," the first provider bid 300 quatloos for the newcomers, meaning the *Enterprise* crew. This episode featured three glowing disembodied brains (I'm not making this up!), sitting on a domed table, who basically spend all their time making wagers about the outcome of battles between various alien species, our intrepid band of explorers among them. Hey, what else are you going to do with your time when you no longer have a body and have solved all the really interesting scientific problems?!? *Star Trek* always did seem to have a fondness for disembodied something-or-others . . . Spock's brain, anyone?!?

you reading this?!?), so I won't be going into it here. However, I think it's fair for you to get a glimpse of it.

After you've downloaded DataVision from the URL mentioned earlier in this section, you'll execute either `datavision.bat` or `datavision.sh`, depending on your operating system, and you'll be greeted with the startup splash screen, as shown in Figure 7-1.

Figure 7-1. *The DataVision startup splash screen*

If you click Open an Existing Report, which you can do because the sample report used in this project is one included with the DataVision distribution, you'll get a typical file open dialog box, and then once you select the report, you'll get a pop-up asking for the username and password for the database. The example report uses a database with no username or password, so just clicking the OK button will get you into the report designer itself, which can be seen Figure 7-2.

In this figure, I've opened up a number of the windows that make up the DataVision GUI. The designer portion itself is the window in the upper-left corner. You can drag and drop fields from the Fields dialog box, and you can modify the field's properties on the report via the Field Format dialog box. The Record Sorting dialog box, of course, allows you to determine how the records used to generate the report are sorted. The Table Linker dialog box allows you to visually design joins between tables, and the Report Formula Language dialog box allows you to determine what language you'd like to write your report formulas in. The Report Description dialog box allows you to see and edit metadata about the report, and the SQL Query window shows you the actual SQL query DataVision will use to generate your report.

You can go ahead and run the report to test it by clicking the Report menu, and then Run Report. The resultant display allows you to export the report in various formats.

Integrating DataVision into your own application requires just a few simple lines of code, which we'll see when we get to that part of RePortal. You can also execute reports via a command line if you wish, and you can have the output go directly to a specified file format, or appear in the Swing viewer, as when you run the report from the designer.

Figure 7-2. *The sample report in the DataVision report designer*

DataVision is a solid reporting package whose price you certainly can't beat: free is pretty tough to beat! Whether it's the best out there I don't know, but I can say for sure that I've personally had great success with it in my own work, and I'm proud to be involved with its continued development.

Now, let's just hope that by observing the outcome of a given report, we aren't altering it,[4] because you never know with all the quantum weirdness around us!

DATAVISION VS. THE COMPETITION

A natural question to ask is whether DataVision is the right choice for you or not outside of RePortal. The answer is, of course, I don't know! Whether you decide to use it or not, it behooves you to explore the options out there and make a decision based on your own comparison.

Some of DataVision's competitors, or perhaps contemporaries is the better term, include JasperReports (`http://jasperforge.org/sf/projects/jasperreports`), Pentaho Reporting, formerly known as JFreeReport (`http://reporting.pentaho.org`), Report Manager for those of you not in the Java world (`http://reportman.sourceforge.net`), RLIB (`http://rlib.sicompos.com`), SpoolTemplate for the PHP crowd (`www.andrioli.com/en/sptpl.html`—notice the LCARS theme to the web site, the perfect tie-in to this chapter), and, of course, Business Objects/Crystal products (`www.businessobjects.com`).

See, I can't be all **that** biased if I mentioned all these!

4. OK, I know it's not *Star Trek*, but still . . . in the *Futurama* (one of the best cartoons ever made) episode "The Luck of The Fryrish," Professor Farnsworth exclaims, "No fair! You changed the output by measuring it!" as a horse race he was watching ended in a "quantum finish." If you got the reference, kudos! If you got the joke, even more kudos!

Quartz

Like reporting, scheduling is something that comes up often enough in the enterprise, and very often it goes hand-in-hand with reporting as a matter of fact. And like with reporting, there's tons of options, ranging from the good ole `cron` command to the Windows Task Scheduler to OpalisRobot (`www.aprompt.com/opalis.htm`). But what about scheduling **within** an application? Oftentimes that means some sort of custom thread-based mechanism, which often works out just fine, but sometimes it's more work than you'd really like.

That's exactly where a product like Quartz from OpenSymphony (`www.opensymphony.com/quartz`) comes into play. Quoting directly from the Quartz home page:

> *Quartz is a full-featured, open source job scheduling system that can be integrated with, or used alongside virtually any J2EE or J2SE application—from the smallest stand-alone application to the largest e-commerce system. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components or EJBs. The Quartz Scheduler includes many enterprise-class features, such as JTA transactions and clustering.*

Quartz makes the job of scheduling tasks in a Java application easy and, one presumes, safer than you probably could yourself (think about all the error handling that has to go into a robust thread-based solution you cook up yourself). With Quartz, scheduling a job is as easy as this:

```
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
JobDetail jobDetail = new JobDetail("jobName", null, MyJob.class);
JobDataMap dataMap = jobDetail.getJobDataMap();
dataMap.put("someData", "someRandomData");
Trigger trigger = TriggerUtils.makeHourlyTrigger();
trigger.setStartTime(new Date());
trigger.setName("MyTrigger");
sched.scheduleJob(jobDetail, trigger);
```

While 10 lines of code probably isn't much in anyone's book, in point of fact, two of those lines are optional (the two dealing with the `dataMap`). In Quartz, everything is done in terms of jobs and triggers. A *job* is the thing that will get executed on some schedule, and a *trigger* is when that job will execute. Here, we begin by getting a new Scheduler instance from the factory and starting it up. Next, we create an instance of `JobDetail` and populate it as appropriate, which means giving it a name (which must be unique within the group), a group (which here is null, which means the job is part of the default group), and telling it which class should be executed. Next is the optional `dataMap` stuff, which is, as the name implies, a map of data that will be passed to the job when it is executed. This is a good way to hand the job resources it needs like handles to JNDI resources and such. Next, a `Trigger` is instantiated. There are numerous variations on a trigger, but the one shown here is a very simple hourly trigger, so this job will fire every hour from the time it's started indefinitely. A trigger has to have a name

just like a job does, and it also needs to have a start time set, which in this case it's simply saying start immediately. Finally, the job is scheduled in the scheduler via the call to `scheduleJob()`, and voilà, we have ourselves a scheduled task without having to mess around with all the thread issues that can come up.

You'll find later on that the code in RePortal is pretty much what's shown here, just a little more around it. We'll also see how to remove jobs from the scheduler, but as you can guess, it's even easier than adding one.

Quartz is a great addition to any project that has scheduling needs, in my opinion. It's one of the best options I've seen in this space, and I've looked at a few over the years.

WAIT, ISN'T THIS SPAWNING THREADS IN A SERVLET CONTAINER?!?

You've probably heard numerous times that spawning threads in a servlet container is a Bad Thing™, and in fact is forbidden by the servlet spec, and you probably are realizing on your own that Quartz must be doing exactly that. So, isn't it an undesirable thing here too?

The answer is, I feel, twofold. First, by using a popular, well-tested library like Quartz, you can feel pretty confident that it's solid, stable, and bug-free. It's likely less dangerous than if you had done it yourself (not to knock your coding abilities, but neither you nor I are Data rerouting a power distribution subnet on a Borg cube, and we tend to make mistakes where Data probably doesn't).

The second fold, as it were, is that the recommendation for not spawning threads is a little on the spurious side in the first place. The recommendation comes about because the container is effectively not in full control of all resources within the container when you spawn threads. It's hard to argue that point, but it's also hard to argue that if you do things right, it's really not a terribly dangerous thing to do. There are ways to ensure the threads behave themselves and don't mess up the container. However, this goes back to the first point: if you do it yourself, it is painfully easy to **not** do everything right and really paint yourself into a nasty corner. If you use something like Quartz, most of the danger goes away.

script.aculo.us

script.aculo.us (`http://script.aculo.us`) is all about the effects! Components that fly onto the page, elements that shrink and expand, parts that fade out of existence, text that color-cycles into existence—all of this can be done with script.aculo.us.

Using script.aculo.us boils down to three simple steps:

1. Import the required JavaScript files.
2. Create a new `Effect` object, passing it the ID of the element to perform the effect on, and optionally parameters for the effect.
3. Sit back and enjoy!

ARE EFFECTS JUST THE DEANNA TROI OF THE ENTERPRISE BRIDGE CREW? (READ: JUST FOR LOOKS)

Let's tackle one question that often comes to mind first: why do we need effects at all? Isn't it just a bunch of superfluous eye candy that doesn't serve much purpose other than to make people go "ooh" and "aah"? Well, first off, if you've ever designed an application for someone else, you know that presentation is an important part of the mix. The more people like how your application looks, the more they'll like how it works—whether it works well or not. It's a relative measure. That's the lesser reason though, although one that should not be quickly dismissed.

The much more important reason has do with how we perceive things. Look around you right now. Pick up any object you want and move it somewhere else. Did the object just pop out from the starting point and appear at the new location? No, of course not! It moved smoothly and deliberately from one place to another. Guess what? This is how the world works! And furthermore, this is how our brains are wired to expect things to work. When things don't work that way, it's jarring, it's confusing, and it's frustrating.

People use movement as a visual cue as to what's going on. This is why modern operating systems are beginning to add all sorts of whiz-bang features, like windows collapsing and expanding. They aren't just eye candy. They do, in fact, serve a purpose: they help our brains maintain their focus where it should be and on what interests us.

In a web application, the same is true. If you can slide something out of view and something else into view, it tends to be more pleasant for the users, and more important, helps them be more productive by not making them lose focus for even a small measure of time.

And seriously, how exactly does the ship's shrink manage to always be around on the bridge?!? Is she necessary personnel for running the *Enterprise*? I know, I know, she finally passed the bridge crew test and was promoted to commander in the episode "Thine Own Self," but still, she was always floating around the bridge long before then. I suppose she was on the lookout for space madness or something, who knows.

The required files, `prototype.js`, `scriptaculous.js`, `builder.js`, `effects.js`, `dragdrop.js`, `slider.js`, and `controls.js`, are simply imported like any other external JavaScript files, via `<script>` tags. And in point of fact, not all of these are actually required; it depends on what features of script.aculo.us you actual plan to use. The `scriptaculous.js` and `prototype.js` files will always be required, the others may not be (see the script.aculo.us documentation for details on how to import only the pieces you need, and for how to determine which files you need and don't need).

Once the required files are present on the page, you initiate an effect like this:

```
new Effect.Appear("div1");
```

This will begin an Appear effect, which makes an element fade in over some time period. Assuming we had a `<div>` on the page with the ID `div1`, that's what would be faded in. What's happening here is a new object is being instantiated, namely the `Effect.Appear` object. The first argument to the constructor for an effect is always the ID of the element to operate on or a DOM object reference itself, the second is a required parameter (although most effects do not have required parameters), and the third is a collection of options. The options are, well, optional! You'll get some set of default values if you don't pass in any options.

Most effects share some common options, as summarized in Table 7-1.

Table 7-1. *Some Common script.aculo.us Effect Options*

Option	Description
`duration`	Sets the duration of the effect in seconds, given as a float. Default value is `1.0`.
`fps`	Targets this many frames per second. Default value is `25`. This cannot be set higher than `100`.
`transition`	Sets a function that modifies the current point of the animation, which is between `0` and `1`. The following transitions are supplied: `Effect.Transitions.sinoidal` (default), `Effect.Transitions.linear`, `Effect.Transitions.reverse`, `Effect.Transitions.wobble`, and `Effect.Transitions.flicker`.
`from`	Sets the starting point of the transition, a floating-point value between `0.0` and `1.0`. Default value is `0.0`.
`to`	Sets the end point of the transition, a floating-point value between `0.0` and `1.0`. Default value is `1.0`.
`sync`	Sets whether the effect should render new frames automatically, which it does by default. If `true`, you can render frames manually by calling the `render()` method of an effect.
`queue`	Sets queuing options. When used with a string, this can be `front` or `end` to queue the effect in the global effects queue at the beginning or end, or a queue parameter object that can have `{position:"front/end", scope:"scope", limit:1}`.
`direction`	Sets the direction of the transition. Values can be `top-left`, `top-right`, `bottom-left`, `bottom-right`, or `center` (`center` is the default value). This is applicable only on Grow and Shrink effects.

All the effects also support supplying callback functions in the options. This allows you to perform some function when certain events in the life cycle of an effect occur. Table 7-2 summarizes the possible callbacks.

Table 7-2. *Possible script.aculo.us Callbacks for Effects*

Callback Event	Description
`beforeStart`	Called before the main effects-rendering loop is started
`beforeUpdate`	Called on each iteration of the effects-rendering loop, before the redraw takes place
`afterUpdate`	Called on each iteration of the effects-rendering loop, after the redraw takes place
`afterFinish`	Called after the last redraw of the effect was made

Here are a few more miscellaneous notes about script.aculo.us effects:

- All of the effects are time-based, which means that if you want an element to expand into view using the Grow effect, and you want it to take two seconds to do so, the effect will take two seconds, regardless of how fast the browser renders each frame.
- In general, the effects are ignorant of the type of element to which you apply them. You should generally be able to apply effects to just about anything. Now, you'll likely find some exceptions, and that's to be expected due to the variations in CSS interpretation by various browsers. But, for the most part, you'll find it to be true.

- For most of these effects to work, you must specify at least some style attributes in-line with the element; they will not work if specified in an external style sheet. For instance, many of the effects you'll see in this chapter's application won't work if the display attribute isn't specified in-line. This is not exactly a big deal, but the first time you try an effect and find that it doesn't do anything, and you wonder why, remember this point. You'll likely save yourself some time!

You will of course see script.aculo.us in action as we review the code, but this should give you a good overview of what it's all about. I unfortunately couldn't find an effect that looked like the Bajoran wormhole opening, but maybe in the next version, I figure.

A Sample Database to Report Against

To properly test RePortal, we'll need a report to play with, and to have a report to play with, we need a database to report against (there **are** in fact other options with DataVision, like delimited data files, but a database is what RePortal is designed to report against, so let's go with it). We don't need 50 gigaquads,[5] just enough to get the point across will do the trick.

Included in the download package for this book, in the `WEB-INF/src` directory of this application, you'll find a directory named `make_derby_database`. This contains a small Java app that does exactly what that directory name implies: creates the test database. All you need to do, assuming you're on a Windows system, is to execute the `doit.bat` file, and the database will be created in the `dvtestdb` subdirectory. Simply copy this directory to your `<TOMCAT>/bin` directory (which will make it available to all webapps running under Tomcat), and you'll be all set. The sample report found in `WEB-INF/src` is designed to run against this database, so you'll have at least one report to play with right away without having to create any in DataVision.

This database consists of a single table named `dvtesttable`, which has the structure shown in Table 7-3, and it contains data listing a bunch of various people and some details about them. It's roughly an employee database, but as you'll see from the test data, it's **very** roughly that!

Table 7-3. *The Structure of the* `dvtesttable` *in the Sample Report Database*

Field Name	Type	Description
`firstname`	`VARCHAR(50)`	First name of the person
`middlename`	`VARCHAR(50)`	Middle name of the person
`lastname`	`VARCHAR(50)`	Last name of the person

5. A quad is a fictional unit of data storage measurement from the *Star Trek* universe. As with byte, a quad can come in various extended units such as kiloquad and gigaquad. How big is a quad, you ask? No one knows, and in fact the term was created by the *Star Trek: The Next Generation* writers specifically to be anachronistic so that new technologies in the real world would never render the advanced *Star Trek* technology too small. Some have reasoned out, from various references on the show, that a quad is equal to 10^15 units, whatever a "unit" might be. Some say that a unit must be along the lines of a "word" today, in that its size is fixed only by the underlying system architecture. Given that, and given a word size of 1 bit, that roughly equates a quad to a modern-day petabit. In this author's humble (yeah, right!) opinion, that makes a quad too small, but hey, when you're making stuff up, you can do whatever you want!

Field Name	Type	Description
title	VARCHAR(50)	Job title (or rough description) of the person
dob	DATE	Date of birth of the person
age	BIGINT	Age of the person
paygrade	CHAR	The pay grade the person is at
salary	NUMERIC	The person's yearly salary

I'll leave it as an exercise to you, my fine reader, to check out the code behind this class. You'll find it not unlike the code we've seen in previous chapters for creating a Derby database. In addition to creating the database, it also populates it with some test data so the report actually generates some output.

Now that we've had a quick look at some of the support pieces that will go into making RePortal a reality, let's start to tear it apart and see what makes it tick (I tried real hard to come up with a *Star Trek* reference that fit here, but I drew a blank . . . feel free to e-mail me suggestions!).

Dissecting RePortal

One quick note to get started: throughout this chapter, all source code has been scrubbed of (most) comments and (most) log statements, and I've also removed some blank lines where it made sense to do so. This was all done to conserve space since this chapter was getting extremely large . . . not quite as large as the V'Ger[6] cloud, but still rather large, and these simple steps saved a few pages.

With that out of the way, let's get down to business, beginning, as has become custom here in this book, with the directory structure of the application, as shown in Figure 7-3.

You're probably a bit tired of me saying the same thing by this point in the book, but I'll say it once more: this is basically a typical Java webapp structure! We have the usual suspect: the `css` directory, where we find two style sheet files. The `styles.css`, like in most of the apps we've seen thus far, is the main style sheet for this application. The `lightbox.css` file contains styles for the pop-up lightbox, which you'll learn all about a bit later, so if you're not familiar with the term, no worries . . . <YodaVoice>you will be . . . YOU . . . WILL . . . BE</YodaVoice>.

The `js` directory contains all our JavaScript files, including the seven script.aculo.us files that were discussed earlier, and the `RePortal.js` file, which is the main JavaScript for the application, and `lightbox.js`, which contains the code for the lightbox pop-up.

In the root, you'll find a single `index.jsp` file that contains all of the markup for the application. Under `WEB-INF`, we find three files: `web.xml`, of course, plus `dwr.xml` and `appConfig.xml`, the last of which is the application-specific configuration for RePortal.

---

6. In *Star Trek: The Motion Picture*, the V'Ger cloud measured 82AUs in diameter . . . well, it did depending on which cut of the movie you saw! In the original 1979 theatrical release, 82AUs is the size given, but it was later realized that 7.626 billion miles was way too large for the *Enterprise* to be able to travel to the heart of it at impulse speeds in the time frame depicted in the movie. So for the later director's cut, the dialog is changed to state the cloud's size as 2AUs (which is still a huge number: 186 million miles—but that's a comparative walk around the block!).

Figure 7-3. *The directory structure of the application*

Under `WEB-INF` we also see the `classes` directory, which contains three files: two files for configuring Commons Logging, and `spring-beans.xml`, which is the configuration file that tells Spring how to instantiate the beans we'll be using.

Beyond that is the `src` directory under `WEB-INF`, which contains all the source code for the project. You'll find the usual directory structure you've seen in the other projects, and all the support files. You'll also find here the `make_derby_database` directory, which contains the Java app to create the sample database the sample report works against (as well as the sample report itself being found here).

With the directory structure out of the way, let's get a quick glimpse of RePortal in action. Naturally enough, seeing it live on your screen is better (it's in full Technicolor!), but if you're

reading this on a plane right now without your laptop, I'll do you the favor and give you a glimpse here. In Figure 7-4, you can see RePortal as it appears when you first access it, assuming you haven't previously logged in and still have an active session.

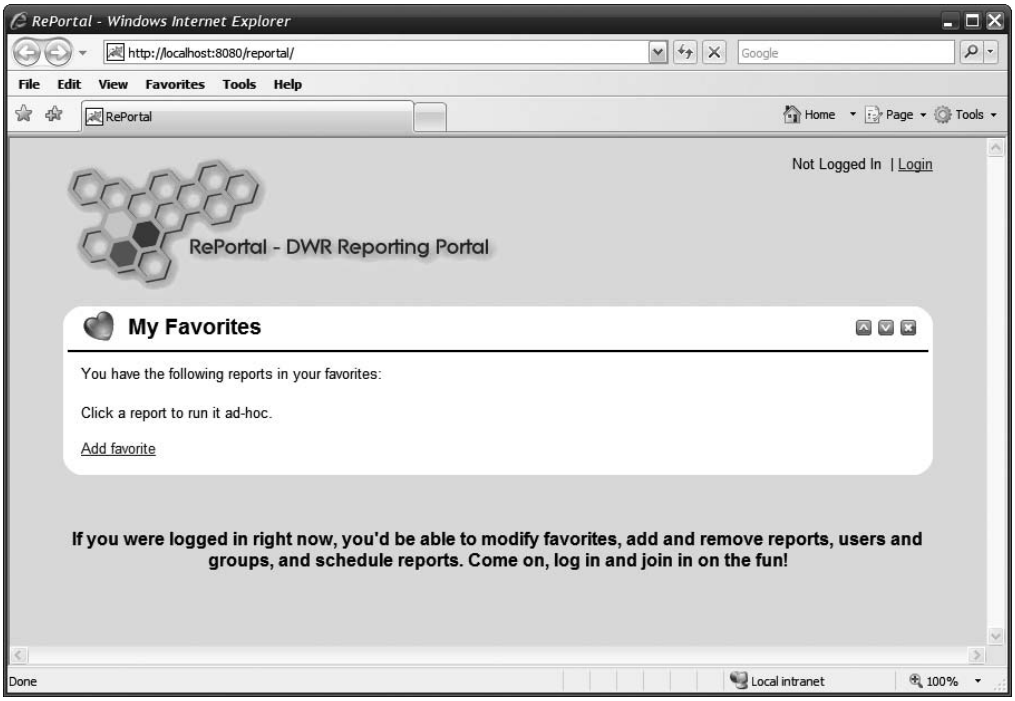

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
Google
File Edit View Favorites Tools Help
RePortal
Home Page Tools
Not Logged In | Login
RePortal - DWR Reporting Portal
My Favorites
You have the following reports in your favorites:
Click a report to run it ad-hoc.
Add favorite
If you were logged in right now, you'd be able to modify favorites, add and remove reports, users and groups, and schedule reports. Come on, log in and join in on the fun!
Done
Local intranet
100%

Figure 7-4. *In keeping with the* Star Trek *theme, here's the first phaser volley: a first look at RePortal*

In the upper-right corner is a link you can use to log in if you have a user account. This opens up all the other functionality RePortal provides. By default, users can see reports made available to all users listed in the favorites area and can run them ad hoc, but they will not be able to add any favorites.

The first time you run the application, the database will be automatically created, and along with it a default administrative account with the username "admin" and the password "admin." It would of course be wise to delete this account **after** creating another for yourself, for security reasons.

Once the user logs in, all the other functionality of RePortal becomes available, and you can see this in Figure 7-5 and Figure 7-6.

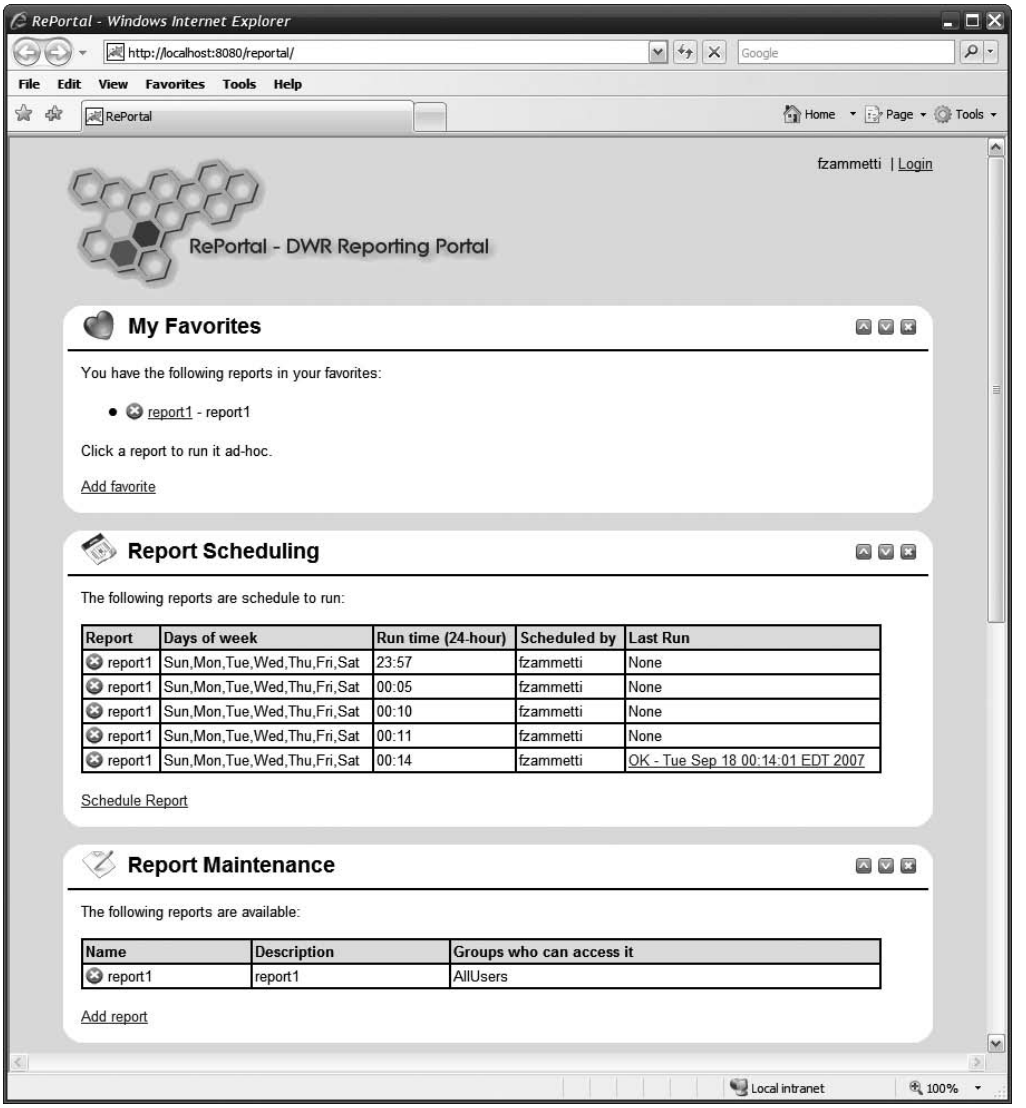

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
fzammetti | Login
RePortal - DWR Reporting Portal
My Favorites
You have the following reports in your favorites:
report1 - report1
Click a report to run it ad-hoc.
Add favorite
Report Scheduling
The following reports are schedule to run:
Report | Days of week | Run time (24-hour) | Scheduled by | Last Run
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 23:57 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:05 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:10 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:11 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:14 | fzammetti | OK - Tue Sep 18 00:14:01 EDT 2007
Schedule Report
Report Maintenance
The following reports are available:
Name | Description | Groups who can access it
report1 | report1 | AllUsers
Add report
Local intranet
100%

Figure 7-5. *This is the top half of the page after you're logged in (picture this figure and the next stitched together vertically—that's the entirety of the RePortal view).*

As you can see, reports can be added to the portal or removed from the portal in the Report Maintenance section. Reports can be scheduled in the Report Scheduling section. Users and groups can be added and removed in the User Administration and Group Administration sections accordingly. Users can modify and store their favorites as well in the My Favorites section.

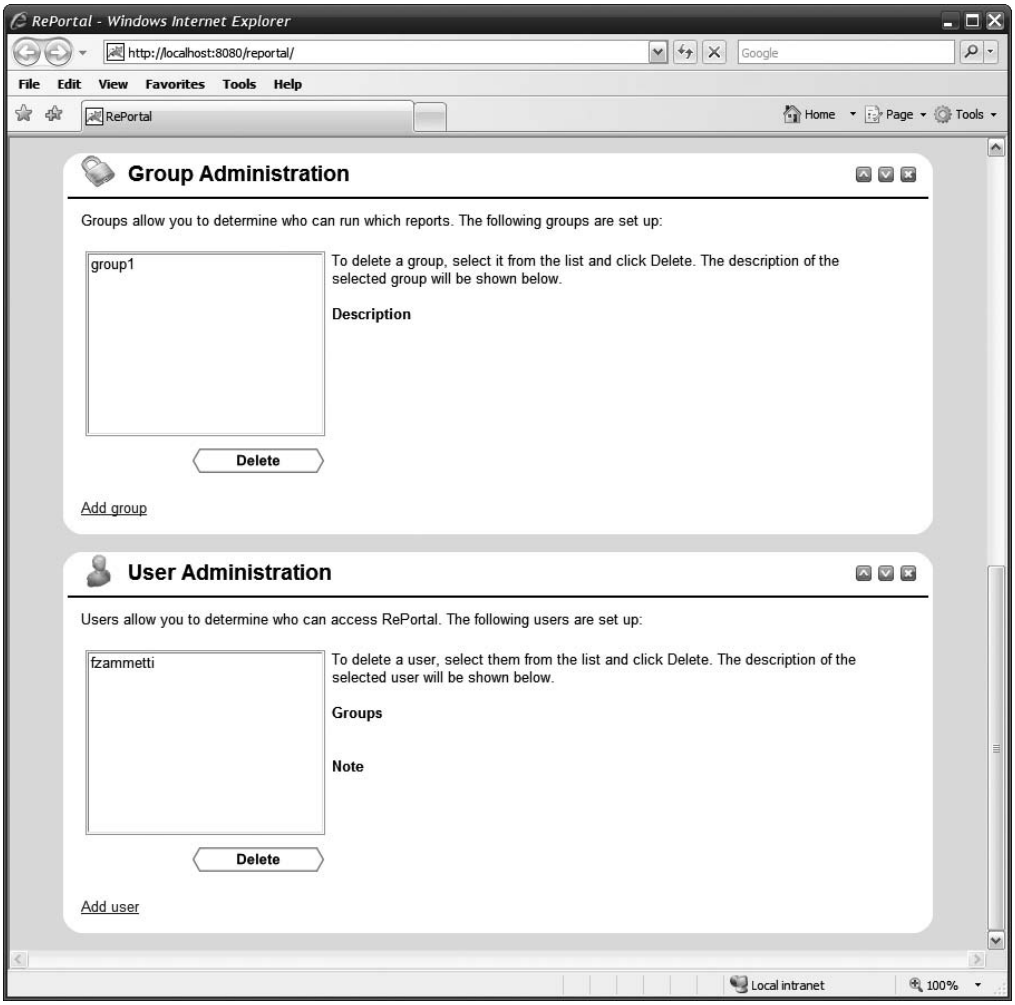

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
Group Administration
Groups allow you to determine who can run which reports. The following groups are set up:
group1
To delete a group, select it from the list and click Delete. The description of the selected group will be shown below.
Description
Delete
Add group
User Administration
Users allow you to determine who can access RePortal. The following users are set up:
fzammetti
To delete a user, select them from the list and click Delete. The description of the selected user will be shown below.
Groups
Note
Delete
Add user
Local intranet
100%

Figure 7-6. *This is the second half.*

In addition, each of these sections can be collapsed to take up only the space of its header section (where the title and three window control buttons appear), and they can be closed entirely if the user wishes to do so. Each of the sections has a link that expands extra functionality, such as adding a report or a user. All of this is done using script.aculo.us effects to give it a little flare.

Now it's time to move on to the actual code, starting with the configuration files. So, get your bio-neural gel packs[7] ready and let's get to it!

---

7. In *Star Trek: Voyager*, the starship *Voyager* is equipped with bio-neural gel packs, a partly biological computer system that supplements the isolinear chip system found in all other federation starship computers. These gel packs are meant to increase processing speed and better organize data. They have the unfortunate tendency to literally get sick from time to time, as happened in at least one *Voyager* episode, "Learning Curve."

Configuration Files

There are four configuration files that go into making RePortal work, three of them in `WEB-INF`, and one in `WEB-INF/classes`. Most of this you've seen already, so I won't go into a ton of detail, but let's go over them at least briefly nonetheless.

web.xml

The `web.xml` file can be seen in Listing 7-1. To say it's nothing exciting would be an understatement of Dyson Sphere–like[8] proportions.

Listing 7-1. *The* `web.xml` *File of RePortal*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app id="reportal" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>RePortal</display-name>

  <listener>
    <listener-class>com.apress.dwrprojects.reportal.ContextListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
    </init-param>
  </servlet>
```

8. In the *Next Generation* episode "Relics" (the episode in which Montgomery Scott from the original series is discovered suspended in the transporter system of a derelict starship), the *Enterprise* discovers a Dyson Sphere, which is basically a giant metallic shell around a star that is used to capture and utilize the entire energy output of said star, plus give a tremendous amount of living space to those who built it, since a planet is no longer necessary. This is similar to the Halo structure from the game of the same name on the Xbox systems, but Halo is considerably smaller than a Dyson Sphere. As is usually the case, the Dyson Sphere seen in *Star Trek* takes some creative license with the real concept: scientist Freeman Dyson describes a Dyson Sphere as a series of satellites orbiting a star, a large number of satellites that is, which in a sense forms a shell, but not a solid one as seen in this episode. The satellites would capture the energy output of the star and transfer it to their creators.

```
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

There is quite literally nothing new to see here, so let's simply move on and get to more interesting things.

appConfig.xml

The `appConfig.xml` file contains two general pieces of information. First is the connection information for the RePortal database itself, which is encoded in the first four elements under the `appConfig` root element shown in Listing 7-2. These represent your typical connection information for a JDBC database, so I won't go into any detail here.

Listing 7-2. *The RePortal-Specific Application Configuration File*

```
<appConfig>

  <databaseDriver>org.apache.derby.jdbc.EmbeddedDriver</databaseDriver>
  <databaseURI>jdbc:derby:reportal;create=true</databaseURI>
  <databaseUsername></databaseUsername>
  <databasePassword></databasePassword>

  <dataSources>
    <dataSource>
      <name>dvtest</name>
      <description>Derby data source for test report</description>
      <databaseDriver>org.apache.derby.jdbc.EmbeddedDriver</databaseDriver>
      <databaseURI>jdbc:derby:dvtestdb</databaseURI>
      <databaseUsername></databaseUsername>
      <databasePassword></databasePassword>
    </dataSource>
  </dataSources>

</appConfig>
```

The second group of information it contains is the `dataSources` element and the child `dataSource` elements it can contain. There can be multiple `dataSource` elements, and each one represents a data source a report can report against. Here, we can see a single data source, named `dvtest`, is configured for use with the sample report. Since there is no hard coupling between a report and a data source, this means that a given report can run against multiple data sources, assuming they have the same structure and naming. Think of being able to run a report against a production database and also against a QA or test database.

Also note that I didn't include maintenance of data sources in the UI. This is partially due to the fact that I couldn't include every feature I would have liked to for two reasons: first, this chapter could have turned into its own book. Second, allowing the addition of data sources as well as reports by a user could open up the possibility of that user getting at data he or she shouldn't be able to access, and at least this way some administrator type has to manually add the appropriate data sources, theoretically making it a little more secure.

dwr.xml

The `dwr.xml` configuration file for RePortal also harbors no real surprises, except for the Spring integration, so have a look at Listing 7-3 and we'll talk about that integration.

Listing 7-3. *Telling DWR All About the "Stuff" (Highly Technical Term) RePortal Needs*

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>

    <convert converter="date" match="java.sql.Time" />
    <convert match="java.lang.Exception" converter="exception" />
    <convert converter="bean" match="java.lang.StackTraceElement"/>
    <convert converter="bean"
      match="com.apress.dwrprojects.reportal.UserDescriptor" />
    <convert converter="bean"
      match="com.apress.dwrprojects.reportal.GroupDescriptor" />
    <convert converter="bean"
      match="com.apress.dwrprojects.reportal.ReportDescriptor" />
    <convert converter="bean"
      match="com.apress.dwrprojects.reportal.ReportScheduleDescriptor" />
    <convert converter="bean"
      match="com.apress.dwrprojects.reportal.DataSourceDescriptor" />

    <create creator="spring" javascript="UserWorker">
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="userWorker" />
    </create>
    <create creator="spring" javascript="GroupWorker">
```

```
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="groupWorker" />
    </create>
    <create creator="spring" javascript="ReportWorker">
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="reportWorker" />
    </create>
    <create creator="spring" javascript="FavoritesWorker">
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="favoritesWorker" />
    </create>
    <create creator="spring" javascript="ReportRunner">
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="reportRunner" />
    </create>
    <create creator="spring" javascript="ReportSchedulingWorker">
      <param name="location" value="spring-beans.xml" />
      <param name="beanName" value="reportSchedulingWorker" />
    </create>

  </allow>
</dwr>
```

For each class that will be instantiated via Spring, we have only to specify its creator as `spring`, add a `location` parameter, and point it at our `spring-beans.xml` file, and that's it! DWR will now dutifully asks Spring for instances of each of the classes when they are remoted.

This isn't the only way to use Spring with DWR, and in fact you may not like the idea of specifying the location of the Spring configuration file for each bean. The alternative involves using a context listener that is part of the Spring MVC package. You can also programmatically tell DWR which Spring BeanFactory instance to use; for instance, if you wrote your own context listener to initialize it, you could then tell DWR to use it. If you prefer either of these approaches, the DWR documentation explains the details here: `http://getahead.org/dwr/server/spring`. In case you're wondering, the only reason I chose to do it the way I did was to avoid the additional configuration and JAR dependency of the listener approach (which is the way I would have done it otherwise). I'm not sure any approach is better than another, although arguably the way it's done in this application is a little harder to maintain, but that's what global search and replace is for, and being it's a single file to do that in, I don't think it's too big a deal.

---

spring-beans.xml

Telling DWR about the Spring configuration file is only part of the equation. We also need to tell Spring how to instantiate our beans, and that's where the `spring-beans.xml` file comes into play, as shown in Listing 7-4.

Listing 7-4. *Getting Spring Ready to Serve Us*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <bean id="databaseWorker"
    class="com.apress.dwrprojects.reportal.DatabaseWorker">
  </bean>
  <bean id="userWorker"
    class="com.apress.dwrprojects.reportal.UserWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>
  <bean id="groupWorker"
    class="com.apress.dwrprojects.reportal.GroupWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>
  <bean id="reportWorker"
    class="com.apress.dwrprojects.reportal.ReportWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>
  <bean id="favoritesWorker"
    class="com.apress.dwrprojects.reportal.FavoritesWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>
  <bean id="reportRunner"
    class="com.apress.dwrprojects.reportal.ReportRunner">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>
  <bean id="reportSchedulingWorker"
    class="com.apress.dwrprojects.reportal.ReportSchedulingWorker">
    <property name="databaseWorker" ref="databaseWorker" />
  </bean>

</beans>
```

You already saw in the introduction to Spring IoC earlier what this configuration file looks like, so there are no surprises here. The one interesting thing though is that each of the beans references another, namely the bean with the ID `databaseWorker`. What happens here is that when, say, the `userWorker` bean is called by DWR, it will ask Spring to instantiate it. Spring will look at this configuration data and see that the `userWorker` bean references the `databaseWorker` bean. So, Spring will instantiate a `databaseWorker` bean, and then inject it into the `userWorker` bean via setter injection. So, when DWR finally gets ahold of the `userWorker` instance, it has the `databaseWorker` it needs too, no coding required. That's pretty neat, no?

Like DWR, Spring can be configured with annotations as well (and don't worry, the next two chapters will be using annotations for DWR). You can get details about this in the Spring documentation, and here's a link to a good starting point: `http://static.springframework.org/spring/docs/2.1.0-m4/reference/beans.html#beans-annotation-config`.

The RePortal Database

We briefly looked at the test database the sample report runs against earlier, but there's a whole other database involved in RePortal, and that's the application database itself. This is more complex than the test database, but isn't very complex overall. In Figure 7-7, you can see the schema for the database in its entirety.

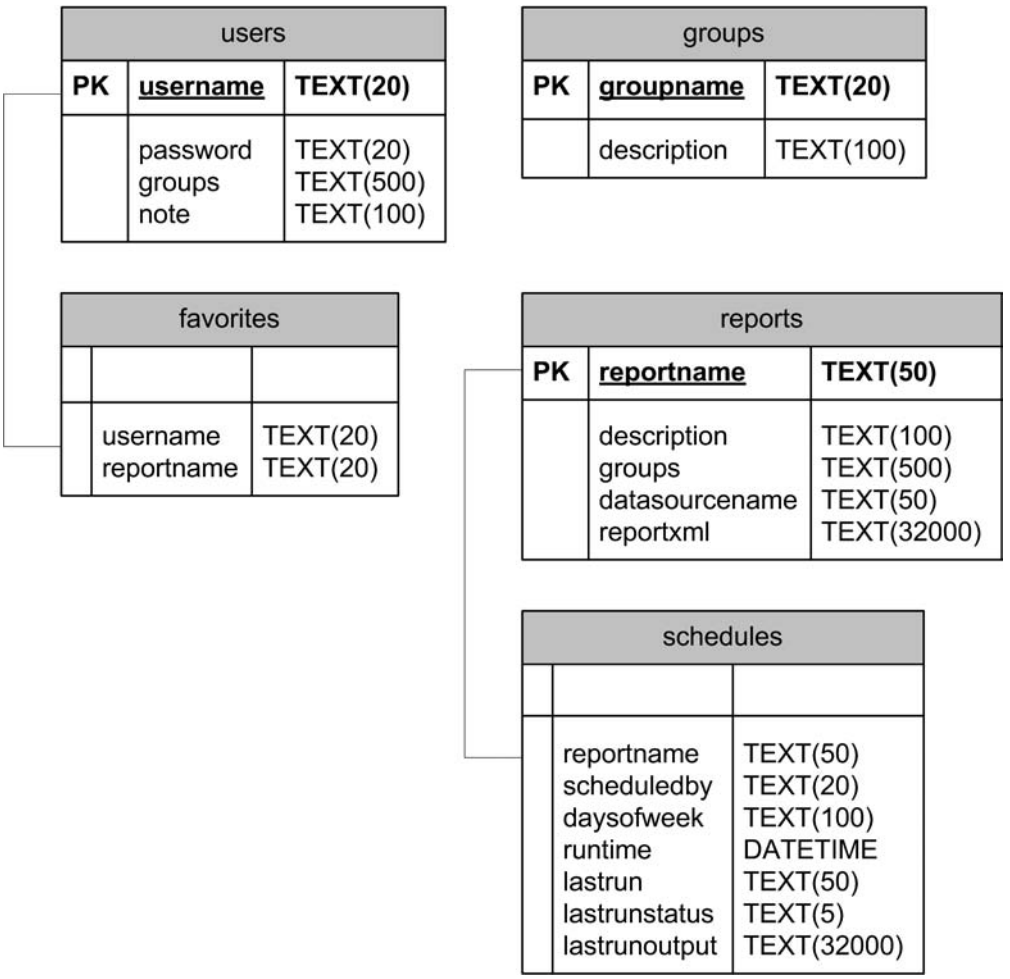


Figure 7-7. *The RePortal database diagrammed*

As you can see, there is a grand total of five tables. The `users` table stores data about users who can access RePortal, including their `username` (which must be unique), `password`, what group(s) they are members of, and a free-form `note` about them. The `groups` field is a comma-separated list of group names (which is taken from the `groupname` field in the `groups` table).

Speaking of the `groups` table, you can see that here as well, and you can see that each group can have a `description`, which is useful in the UI, in addition to its name (which must be unique).

The `favorites` table stores favorites for a given user, and you can see there is a relationship between them on the `username` field (this is a "soft" relationship, i.e., there is no database constraint or anything in place to enforce this). Each entry in this table stores the name of the user in the `username` field and the name of the report in the `reportname` field (which maps to the `reportname` field in the reports table).

And moving right along is the `reports` table itself. Here you can see that we store the name of the report in the `reportname` field, a `description` of the report, the list of group(s) that can access the reports (another comma-separated list), as well as the name of the data source the report will use and the XML that makes up the report. Note that because the `reportname` field must be unique, that implies that if you have the same report that runs against different data sources, you will have to come up with a different name for each instance.

Finally, the `schedules` table stores information about scheduled report jobs. This includes the name of the report (`reportname`), who scheduled it (which maps to the `username` field of the users table), what days of the week it should run (a comma-separated list of day abbreviations), the time the report should run, the date/time the job last ran, the outcome of that run in the `lastrunstatus` field, and the actual HTML output of the DataVision report in `lastrunoutput`.

It's a simple database, but it serves our purposes just fine. By the way, the database will be automatically created via the application code as you'll see later, so no need to do anything special to get the application up and running. As noted previously, an administrative user will also be created as part of the database creation, so you should be able to log in to the application immediately using the admin/admin username/password combination.

The Client-Side Code

Having seen all the configuration files and gotten an overview of the database the application will use, it's time to start looking at the actually code, beginning with our style sheets.

styles.css

Because it's a little on the long side, the `styles.css` file is not shown here. Please take a look at it from the download bundle as I discuss just one or two interesting points here (the rest of it is pretty standard stuff that's you've seen before, at least in the projects in this book).

You'll find a couple of interesting bits with regard to the rounded corners of the sections of the portal. This is accomplished with a couple of images for the corners that are set as backgrounds of some container elements. There are quite a few ways to implement rounded corners; some require images, while others do not. I'm not here to advocate one approach over another; I just came up with an implementation that works well enough.

Another interesting note is in the `cssSectionDivider` class. You'll notice I use the underscore-in-the-name trick to have a different setting for Internet Explorer vs. other browsers

(other browsers will ignore the attribute with the underscore in its name, but IE will ignore the underscore, so the attribute with the underscore in its name effectively overrides where the same attribute was set just above it).

A Beacon in the Night: lightbox.css and lightbox.js

You've seen me use the term "lightbox" a few times already, and you may know what it is, but if not, let me explain. The lightbox effect is a product of the Web 2.0 movement (it may have existed before, but it's been popularized only quite recently). The basic idea is that you take a web page, and you "dim it out," and then put some sort of pop-up over it that is lighter than the dimmed page—hence the name "lightbox." In Figure 7-8, you can see the lightbox in action. This figure shows what you see when you click the login link.

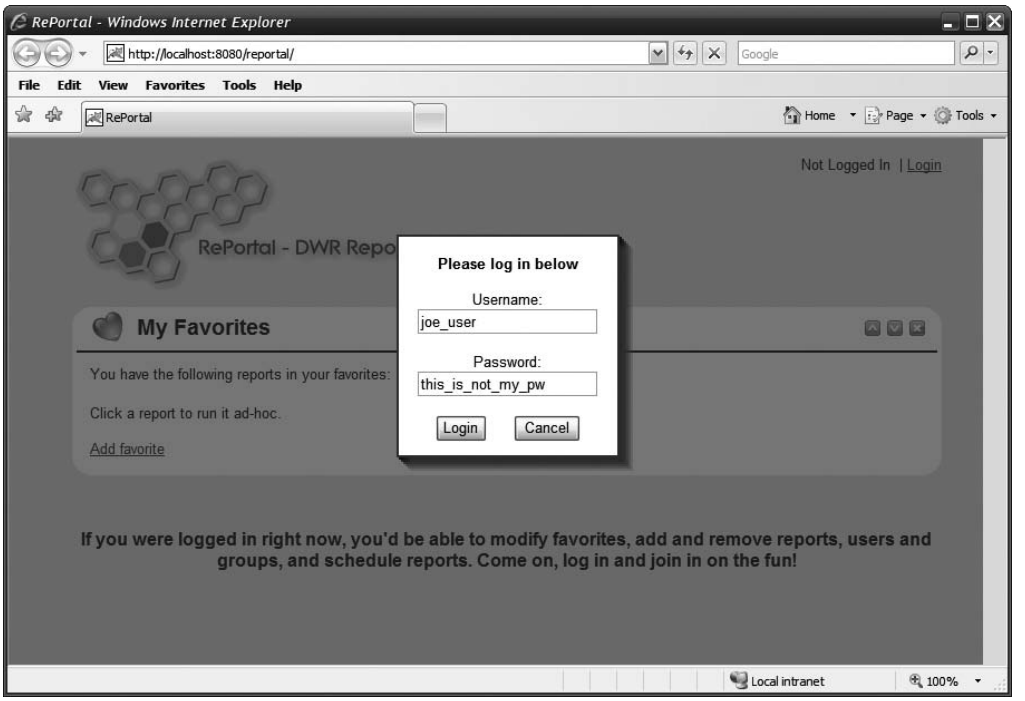

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
Not Logged In | Login
RePortal - DWR Repo
Please log in below
Username:
joe_user
Password:
this_is_not_my_pw
Login
Cancel
My Favorites
You have the following reports in your favorites:
Click a report to run it ad-hoc.
Add favorite
If you were logged in right now, you'd be able to modify favorites, add and remove reports, users and groups, and schedule reports. Come on, log in and join in on the fun!
Local intranet
100%

Figure 7-8. *The lightbox in action: logging in*

There are more lightbox implementations available today than you can shake a Ferengi energy whip at, but I couldn't find one that quite suited me (a lightbox I mean, I have a perfectly good energy whip already). So, I set about to implement my own. Fortunately, the basic concept is quite simple, but the devil, as usual, is in the details.

It all begins with some CSS magic, which is on display in Listing 7-5. The real key to it all is three settings: `-moz-opacity`, `opacity`, and `filter:alpha(opacity=xx)`. The first two set opacity in Firefox, Netscape, and other such browsers, and the last one sets it in Internet Explorer.

Listing 7-5. *The First Half of the Lightbox Equation: the Style Sheet*

```
.cssLightboxBlocker {
  display          : none;
  position         : absolute;
  top              : 0px;
  left             : 0px;
  width            : 100%;
  height           : 5000px;
  z-index          : 9997;
  background-color : #404040;
  -moz-opacity     : 0.6;
  opacity          : .60;
  filter           : alpha(opacity=60);
}

.cssLightboxPopup {
  border           : 2px solid #ff0000;
  background-color : #ffffff;
  z-index          : 9999;
  position         : absolute;
}

.cssLightboxShadow0 {
  display          : none;
  position         : absolute;
  background-color : #000000;
  z-index          : 9998;
  filter           : alpha(opacity=80);
  opacity          : .80;
  -moz-opacity     : 0.8;
}
.cssLightboxShadow1 {
  display          : none;
  position         : absolute;
  background-color : #101010;
  z-index          : 9998;
  filter           : alpha(opacity=70);
  opacity          : .70;
  -moz-opacity     : 0.7;
}
.cssLightboxShadow2 {
  display          : none;
  position         : absolute;
  background-color : #202020;
  z-index          : 9998;
  filter           : alpha(opacity=60);
  opacity          : .60;
```

```
  -moz-opacity     : 0.6;
}
.cssLightboxShadow3 {
  display          : none;
  position         : absolute;
  background-color : #303030;
  z-index          : 9998;
  filter           : alpha(opacity=50);
  opacity          : .50;
  -moz-opacity     : 0.5
}
.cssLightboxShadow4 {
  display          : none;
  position         : absolute;
  background-color : #404040;
  z-index          : 9998;
  filter           : alpha(opacity=40);
  opacity          : .40;
  -moz-opacity     : 0.4;
}
.cssLightboxShadow5 {
  display          : none;
  position         : absolute;
  background-color : #505050;
  z-index          : 9998;
  filter           : alpha(opacity=30);
  opacity          : .30;
  -moz-opacity     : 0.3;
}
```

The way a lightbox works is this . . . first, you have an image, or a `<div>`, that covers the entire page. It's typically a grayish color, but could be anything the designer desires. This element is positioned absolutely and sized so it'll cover the entire page (at least the displayed potion of the page, i.e., sized to the browser window). When it's time to show the lightbox, this "blocker" element, as I like to call it, is shown. It has opacity set to something between opaque and transparent. This is what gives the effect of the page being dimmed. It has another very important job: because its `z-index` is set to something very high, it floats on top of everything else on the page, effectively blocking user input to anything other than the lightbox pop-up (see why I like the term "blocker"?).

At the same time, the pop-up is shown, and this has a slightly higher `z-index` than even the blocker, which allows it to still be functional when the rest of the page is not. That is, in a nutshell, how it works.

Now, when you look at this style sheet, the `cssLightboxBlocker` class styles that blocker element, which is a `<div>` in my implementation. You'll notice that the width is 100%, and the height is 5000 pixels. This should allow it to effectively block pages being displayed in any browser other than the room-sized display in the astrometrics lab on Voyager, regardless of how vertically large the page is. You can see the opacity set here to 60% (`.6` and `.60` meaning

the same, but it's specified differently for those attributes than the `filter` attribute). This is exactly what we need to make the effect work.

The `cssLightboxPopup` is the style applied to the `<div>` that contains the pop-up itself. As you can see, its `z-index` is a little higher than the blocker style, so that will put it above the blocker so it will remain active. Notice that its position isn't specified. In the JavaScript behind the lightbox, which is shown in Listing 7-6, the pop-up is centered automatically, so no need to specify a position that'll just be overridden anyway.

The remaining styles deal with the drop shadow that is seen below and to the right of the lightbox pop-up, but that takes a little explanation that will make a bit more sense after seeing the JavaScript, so let's get to that now.

Listing 7-6. *The Second Half of the Lightbox Equation: the JavaScript*

```
function lightboxPopup(inPopupID, inPopupVisible) {

  if (inPopupVisible) {

    var popupDiv = document.getElementById(inPopupID);
    document.getElementById("divLightboxBlocker").style.display = "block";
    var st = document.body.scrollTop;
    document.body.style.overflow = "hidden";
    document.body.scrollTop = st;

    var lca = null;
    var lcb = null;
    var lcx = null;
    var lcy = null;
    var iebody = null;
    var dsoctop = null;

    popupDiv.style.display = "block";

    if (window.innerHeight) {
      lca = window.innerHeight;
    } else {
      lca = document.body.clientHeight;
    }
    lcb = popupDiv.offsetHeight;
    lcy = (Math.round(lca / 2)) - (Math.round(lcb / 2));
    iebody = (document.compatMode &&
      document.compatMode != "BackCompat") ?
      document.documentElement : document.body;
    dsoctop = document.all ? iebody.scrollTop : window.pageYOffset;
    popupDiv.style.top = ((lcy + dsoctop) - 50) + "px";

    if (window.innerWidth) {
      lca = window.innerWidth;
    } else {
```

```
      lca = document.body.clientWidth;
    }
    lcb = popupDiv.offsetWidth;
    lcx = (Math.round(lca / 2)) - (Math.round(lcb / 2));
    iebody = (document.compatMode &&
      document.compatMode != "BackCompat") ?
      document.documentElement : document.body;
    dsocleft = document.all ? iebody.scrollLeft : window.pageXOffset;
    popupDiv.style.left = lcx + dsocleft + "px";

    for (var i = 0; i < 6; i++) {
      document.getElementById(inPopupID + "_Shadow" + i).style.display =
        "block";
    }
    var xyAdjust = 12;
    for (var i = 5; i >= 0; i--) {
      document.getElementById(inPopupID + "_Shadow" + i).style.left =
        lcx + xyAdjust + dsocleft + "px";
      document.getElementById(inPopupID + "_Shadow" + i).style.top =
        (((lcy + dsoctop) - 50) + xyAdjust) + "px";
      xyAdjust -= 2;
    }

  } else {

    document.getElementById(inPopupID).style.display = "none";
    document.getElementById("divLightboxBlocker").style.display = "none";
    document.body.style.overflow = "";
    for (var i = 0; i < 6; i++) {
      document.getElementById(inPopupID + "_Shadow" + i).style.display = "none";
    }

  }

}
```

You'll notice the branching logic right off the bat. This is so that the same method can be called to both show and hide a lightbox pop-up; it's just a question of whether the `inPopupVisible` parameter is `true` (show pop-up) or `false` (hide pop-up). The `inPopupID` parameter specifies the DOM ID of the `<div>` containing the pop-up contents by the way.

The first thing done here is to get a reference to the pop-up `<div>`. Next, the blocker element is shown by modifying its `display` style attribute. After that comes something that's fairly unique to my implementation: the current amount the page is scrolled is stored, and the `overflow` attribute of the body element is set to hide content outside the display area. This has the effect of hiding the scrollbars. The reason for this is that when the lightbox pop-up is shown, the user can still scroll the page, which means scrolling the pop-up out of view. This kind of annoyed me to be honest, so I set about to solve it, and came up with this solution.

There's of course a problem, and that is in Firefox, when you change the `overflow` attribute like this, it has the unfortunate side effect of scrolling the document all the way to the top. So, that's why the `scrollTop` attribute is set to the value retrieved before the `overflow` attribute was changed. To the user, it looks like the page didn't scroll, it simply stayed where it was, which is exactly what you'd want it to do.

After all that comes two blocks of code that you've seen in other projects in this book, and that's the code to center the pop-up. I'll skip going over these again at this point in time.

The last step of showing the lightbox is to deal with the shadow. Now, you might initially think that just using a static image of a shadow would work, but in fact it won't because Internet Explorer's support for PNG opacity is pitiful at best. So, an image-based shadow wouldn't be possible (well, I probably shouldn't go **that** far as I'm sure someone smarter than me has figured it out . . . maybe Dax, in a former host?), but in any case, it's pretty tough to do and have it look decent. But, there's a fairly easy cheat, and that's to use `<div>`s with a proper setup.

That proper setup boils down to creating a series of `<div>`s, each with a progressively lighter appearance (both opacity and color are used) going out from the lightbox. If these `<div>`s have the same dimensions as the lightbox and are positioned moving away from the lightbox right and down, you get a drop shadow effect. Plus, with the same opacity settings seen on the blocker element, you can get a fairly good blend look to it too.

So, two loops are used. The first one simply shows the shadow `<div>`s. They have to be visible before they can be properly manipulated within the second loop, which comes next. This loop is responsible for positioning the `<div>`s (they are assumed to be sized properly, and in fact are as we'll see in the markup). This loop essentially moves each `<div>` 2 pixels distance for each iteration, and if you look at the styles related to the shadows in the CSS, you'll note that each style gets progressively lighter. The loop places the lightest first, furthest from the pop-up, and then works backwards, placing progressively darker shadows going closer to the pop-up. In this way, each subsequently darker shadow overlays the portion of the previous lighter one that should be obscured to give the proper gradient effect.

The markup for the pop-up is found in `index.jsp`, which we'll be reviewing more fully very soon, but I think this is an appropriate place to see the relevant snippet from that file:

```
<div id="divPleaseWait" class="cssLightboxPopup"
  style="width:140px;height:50px;display:none;">
  <br>
  <div class="cssPleaseWait">Processing...</div>
</div>
<div id="divPleaseWait_Shadow0" class="cssLightboxShadow0"
  style="width:140px;height:50px;"></div>
<div id="divPleaseWait_Shadow1" class="cssLightboxShadow1"
  style="width:140px;height:50px;"></div>
<div id="divPleaseWait_Shadow2" class="cssLightboxShadow2"
  style="width:140px;height:50px;"></div>
<div id="divPleaseWait_Shadow3" class="cssLightboxShadow3"
  style="width:140px;height:50px;"></div>
<div id="divPleaseWait_Shadow4" class="cssLightboxShadow4"
  style="width:140px;height:50px;"></div>
<div id="divPleaseWait_Shadow5" class="cssLightboxShadow5"
  style="width:140px;height:50px;"></div>
```

Note that I chose to show the lightbox pop-up that is seen when a report is executing instead of the login pop-up simply because this version is smaller for print purposes. Rest assured, aside from the actual content being different, the overall structure is identical. Here you can see the `<div>` for the pop-up itself, plus the six `<div>`s for the shadows. Note the size of each shadow matches that of the pop-up `<div>`.

I should also note that the lightbox functionality is completely independent, and you should be able to drop it in another project and use it right away. I purposely did that because frankly I needed this functionality for one of my own projects! Hopefully, you find use for it as well outside this book; it's all ready for you if you do need it.

index.jsp

In the last section, we got a little preview of `index.jsp`, but now it's time to tear into it full steam ahead. This is a fairly large source file though, and you'll find that it's actually fairly repetitive. There is a structure that tends to repeat, by and large, so what I'll do here, after we discuss the truly unique portions, will be to show one example of this standard structure, and you can then review the other instances of it later to get the full picture.

The first section we encounter is this:

```
<%
  String username = (String)session.getAttribute("username");
  if (username == null) {
    username = "";
  }
%>
```

This simply looks for an attribute named `username` in session, and if found, it grabs it. This is then used later when setting up the UI, such that if the user has already logged in, the UI will have all the features enabled; otherwise, only the My Favorites section will be shown.

After that comes the head of the document:

```
<html>
  <head>
    <title>RePortal</title>

      <link rel="StyleSheet" href="css/styles.css" type="text/css">
      <link rel="StyleSheet" href="css/lightbox.css" type="text/css">

      <script src="dwr/engine.js"></script>
      <script src="dwr/util.js"></script>
      <script src="dwr/interface/UserWorker.js"></script>
      <script src="dwr/interface/GroupWorker.js"></script>
      <script src="dwr/interface/ReportWorker.js"></script>
      <script src="dwr/interface/FavoritesWorker.js"></script>
      <script src="dwr/interface/ReportRunner.js"></script>
      <script src="dwr/interface/ReportSchedulingWorker.js"></script>
```

```
    <script src="js/prototype.js" type="text/javascript"></script>
    <script src="js/scriptaculous.js?load=effects"
      type="text/javascript"></script>
    <script src="js/lightbox.js" type="text/javascript"></script>
    <script src="js/RePortalClass.js" type="text/javascript"></script>

  </head>

  <body class="cssBody" onLoad="RePortal.init('<%=username%>');">
```

This is pretty much just the typical almost boilerplate code at this point. The style sheets are imported, as are all the DWR JavaScript files, and then the script.aculo.us file, prototype file, lightbox file, and main code for the application in the `RePortalClass.js` file.

Now, note the import for `scriptaculous.js`. Recall earlier where I said I included all the script.aculo.us files, even though only effects were used? Well, here I've made that even more pointless than a stun setting on a Romulan disruptor by telling script.aculo.us to only load the effects! See, when you import `scriptaculous.js`, it goes off and loads all the other files by default, which is nice, since you only have to remember to import it (and `prototype.js`, which it depends on). However, if you add the `load` request parameter, you can specify exactly which modules you want to load, as I've done here to only load the `effects.js` file.

You'll notice I also showed the `<body>` tag at the end of that code, and the thing to note there is how the username, which we captured earlier, is passed to the `init()` method of the `RePortal` class, which we'll see later, and this is done `onLoad` of the page. If the user wasn't logged in, and hence no `username` attribute was present, an empty string will be passed, and the `init()` method can react accordingly.

The first thing we spot in the `<body>` is the markup for what's seen in the upper-right corner. Here's that portion of `index.jsp`:

```
<table border="0" cellpadding="0" cellspacing="0" width="780"
  align="center"><tr><td>

  <table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr>
      <td valign="top" ><img src="img/logo.gif"></td>
      <td valign="top" align="right">
        <span id="spanUsername">Not Logged In</span>
         | <a href="javascript:void(0);"
          onClick="if(RePortal.username!=null){➥
          alert('You are already logged in');}else{➥
          lightboxPopup('divLogin', true);}">Login</a>
      </td>
    </tr>
  </table>
```

As you can see, there isn't much to it. When the login link is clicked, we do a little logic check to determine whether the user is logged in (the `RePortal` class has a `username` field, which gets set to the value passed in to the `init()` method on page load). If the user isn't logged in, we use the `lightboxPopup()` function seen earlier to display the login pop-up. Otherwise, we inform the user he or she is already logged in.

What we're now going to look at is the markup for a single section of the portal, namely the My Favorites section. There are five different sections on the portal, but I'm only going to show the markup for this one here because these are actually fairly repetitive. Naturally, they aren't identical, but they all follow a similar structure and only really differ in some basic HTML and the JavaScript functions that are called for various events. You definitely should examine the others on your own, but this description will adequately give you the idea of how they all basically work.

```
<div class="cssSectionContainer" id="divMyFavorites">
  <div class="cssSection">
    <div class="cssSectionTop">
      <div>
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
          <tr>
            <td>
              <span class="cssSectionTitle">
                <img align="absmiddle" src="img/icoMyFavorites.gif" >
                   My Favorites
              </span>
            </td>
            <td align="right" class="cssSectionButtons">
              <img src="img/sectionCollapse.gif" class="cssSectionButton"
                onClick="RePortal.collapseSection('divMyFavorites');">
              <img src="img/sectionExpand.gif" class="cssSectionButton"
                onClick="RePortal.expandSection('divMyFavorites');">
              <img src="img/sectionClose.gif" class="cssSectionButton"
                onClick="RePortal.removeSection('divMyFavorites');">
            </td>
          </tr>
        </table>
      </div>
    </div>
    <div class="cssSectionDivider"><hr color="#000000" width="99%"></div>
    <div class="cssSectionContent">
      You have the following reports in your favorites:
      <br>
      <ul id="favorites_favoritesList"></ul>
      Click a report to run it ad-hoc.
      <br><br>
      <a href="javascript:void(0);"
```

```
        onClick="RePortal.showHideAddSection('favorites_addReport');">
        Add favorite
      </a>
      <div id="favorites_addReport" style="display:none;">
        <table border="0" cellpadding="2" cellspacing="2" width="100%">
          <tr>
            <td valign="top" width="1">
              <select size="10" id="favorites_reportsList"
                onChange="RePortal.favoritesDisplayReportInfo(this.value);"
                class="cssSelect">
              </select>
            </td>
            <td valign="top">
              To add a report to your favorites, select it from the list
              and click Add.  The description of the selected report will
              be shown
              below.
              <br><br>
              <b>Description</b>
              <div id="favorites_reportDescription"></div>
            </td>
          </tr>
          <tr>
            <td align="right">
              <img src="img/btnAdd0.gif" class="cssButton"
                onClick="RePortal.addReportToFavorites();"
                onMouseOver="this.src='img/btnAdd1.gif';"
                onMouseOut="this.src='img/btnAdd0.gif';">
            </td>
            <td> </td>
          </tr>
        </table>
      </div>
    </div>
    <div class="cssSectionBottom"><div></div></div>
  </div>
</div>
```

It's all actually pretty standard HTML. You'll note that all of the JavaScript event handlers call methods of the `RePortal` class, and probably all of those methods have names that give you a good idea what they're all about. We'll of course examine them shortly.

One of the things you can see here is the code behind the Expand, Collapse, and Close buttons that are present for each section. These allow the user to control the sections to make more room where appropriate. For instance, in Figure 7-9, you can see how the portal would look if the user were to collapse all the sections.

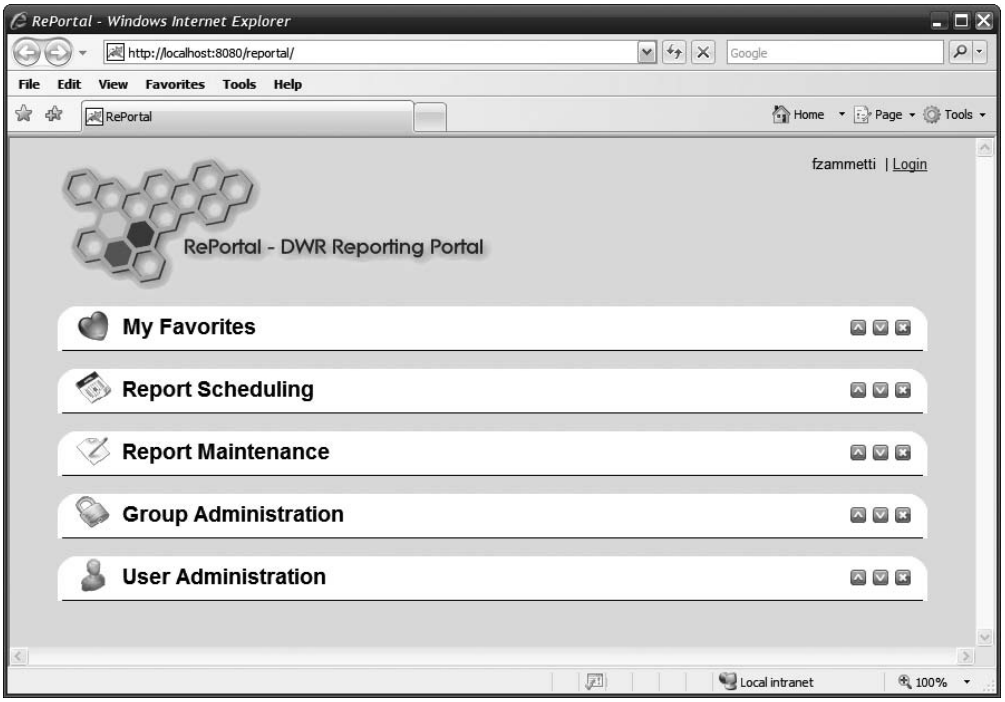


Figure 7-9. *The various sections of RePortal can be expanded/collapsed when needed, as shown here, where all the sections are collapsed (which isn't very useful, but possible nonetheless).*

In Figure 7-10, you can see what it looks like if all but one section is collapsed.

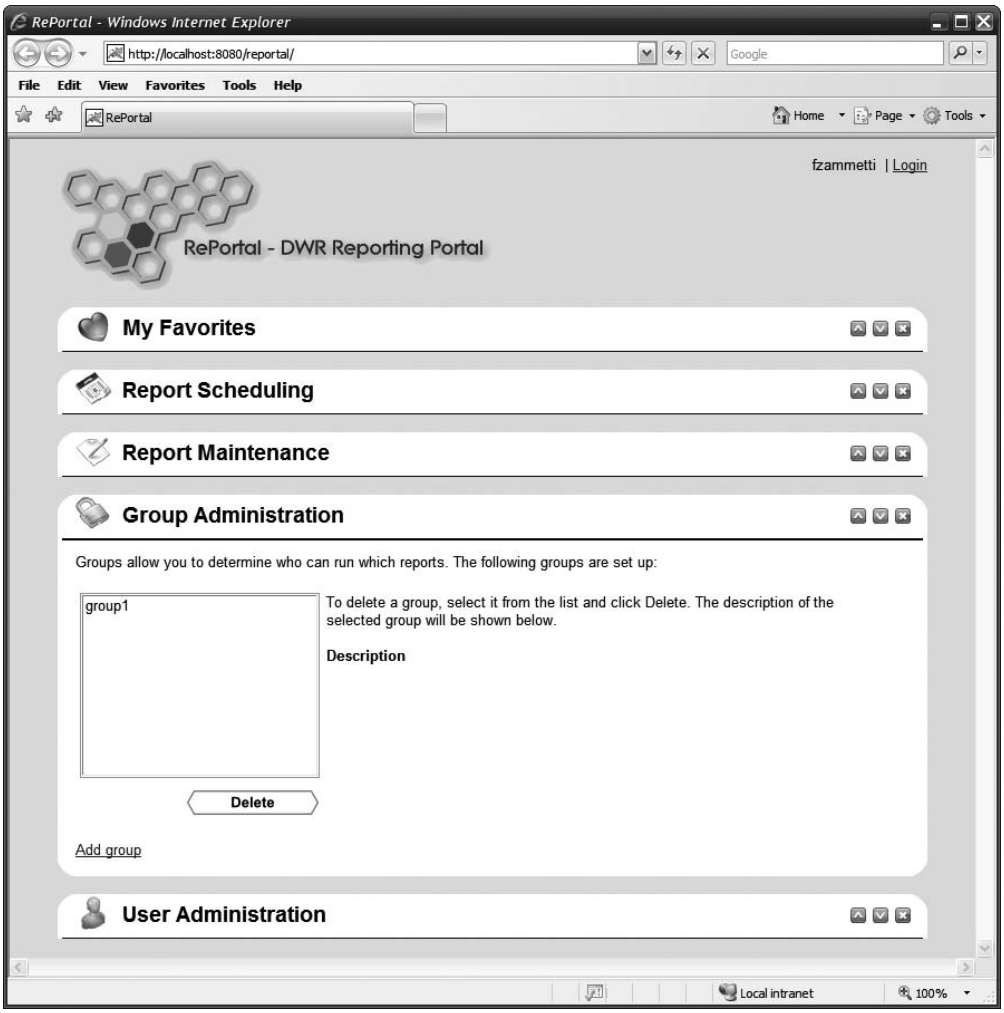

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
fzammetti | Login
RePortal - DWR Reporting Portal
My Favorites
Report Scheduling
Report Maintenance
Group Administration
Groups allow you to determine who can run which reports. The following groups are set up:
group1
To delete a group, select it from the list and click Delete. The description of the selected group will be shown below.
Description
Delete
Add group
User Administration
Local intranet
100%

Figure 7-10. *Here's just one section expanded, which is a bit more useful.*

The next bit of markup is the login lightbox pop-up. We already saw how it's shown; now we'll see how it's formed:

```
<div id="divLogin" class="cssLightboxPopup"
  style="width:200px;height:200px;display:none;">
  <center>
    <br>
    <b>Please log in below</b>
    <br><br>
    Username: <input type="text" size="21" maxlength="20" id="login_username">
    <br><br>
    Password: <input type="password" size="21" maxlength="20" id="login_password">
    <br><br>
```

```
    <input type="button" onClick="RePortal.logUserIn();" value="Login">
        
    <input type="button" value="Cancel"
      onclick="lightboxPopup('divLogin', false);">
  </center>
</div>
<div id="divLogin_Shadow0" class="cssLightboxShadow0"
  style="width:200px;height:200px;"></div>
<div id="divLogin_Shadow1" class="cssLightboxShadow1"
  style="width:200px;height:200px;"></div>
<div id="divLogin_Shadow2" class="cssLightboxShadow2"
  style="width:200px;height:200px;"></div>
<div id="divLogin_Shadow3" class="cssLightboxShadow3"
  style="width:200px;height:200px;"></div>
<div id="divLogin_Shadow4" class="cssLightboxShadow4"
  style="width:200px;height:200px;"></div>
<div id="divLogin_Shadow5" class="cssLightboxShadow5"
  style="width:200px;height:200px;"></div>
```

It's just a `<div>` with the `cssLightboxPopup` style applied. The content is just a couple of HTML form elements, plus some surrounding text. Below that `<div>` you find the six `<div>`s that are the drop-down shadow of the pop-up.

You'll find the "Processing..." pop-up has the same basic structure, just less actual content, of course.

RePortalClass.js

As you examined `index.jsp`, you will have noticed numerous calls to the `RePortal` class, so now we need to look at that code, which is where all the real client-side code of this application really lives.

The init() Method

To begin, we have the `init()` method, which we saw earlier is called when `index.jsp` loads. It is responsible for initializing the application, and it is this code:

```
this.init = function(inUsername) {

  dwr.engine.setErrorHandler(RePortal.exceptionHandler);

  if (inUsername != "") {
    this.username = inUsername;
    dwr.util.setValue("spanUsername", RePortal.username);
    dwr.util.byId("divNotLoggedInMessage").style.display = "none";
    dwr.util.byId("divReportScheduling").style.display = "";
    dwr.util.byId("divReportMaintenance").style.display = "";
    dwr.util.byId("divGroupAdministration").style.display = "";
    dwr.util.byId("divUserAdministration").style.display = "";
  }
```

```
  ReportWorker.getReportsList(
    { callback : RePortal.updateReportListCallback }
  );

  GroupWorker.getGroupsList(
    { callback : RePortal.updateGroupsListCallback }
  );

  UserWorker.getUsersList( { callback : RePortal.updateUserListCallback } );

  FavoritesWorker.getFavoritesForUser(this.username,
    { callback : RePortal.updateFavoritesCallback }
  );

  setInterval("ReportSchedulingWorker.getScheduledReportsList(" +
    "{ callback : RePortal.updateScheduledReportListCallback } );", 15000);
  ReportSchedulingWorker.getScheduledReportsList(
    { callback : RePortal.updateScheduledReportListCallback }
  );

  ReportRunner.getDataSourceList(
    { callback : function(inList) {
        dwr.util.removeAllOptions("reportMaintenance_addDataSource");
        for (var i = 0; i < inList.length; i++) {
          dwr.util.addOptions("reportMaintenance_addDataSource",
            [ inList[i].name ]
          );
        }
      }
    }
  );

}
```

The first thing done is to set an error handler for DWR to use in response to any exceptions thrown by the server. Next, we examine the username passed in. If it is not an empty string, we go ahead and display the username in the upper-right corner using the `dwr.util.setValue()` function. Next, we hide the message that is by default shown, the one telling users they should log in to see all the other features of RePortal. In addition, all the other sections aside from My Favorites are now shown, and the portal is ready to rock and roll for that logged-in user.

Following that work is a series of DWR remote calls to populate various parts of the portal, including the list of favorites for the user, the list of reports known to the portal, the list of scheduled report jobs, the list of users, and the list of groups.

One that bears some explanation is the `getScheduledReportsList()` list call. The list of scheduled report jobs includes the status of the last time the report executed, and that status is clickable to display the report if it was successful. Because these reports might be running as

the user is looking at the portal, it would sure be nice if we could update that display every so often for the user. That's what the `setInterval()` call does. It results in a periodic call (every 15 seconds, or 15000 milliseconds as you see it here) to the `getScheduledReportsList()` method of the `ReportSchedulingWorker` class on the server. This redraws the list of scheduled reports, thereby updating the status for any that may have run.

You'll also notice the data source list update is a bit different from the rest. All of the other lists that are populated by the calls here will need to be called at various times later; for instance, when a user is added, the list of users needs to be updated, so another call to `UserWorker.getUsersList()` will be required. However, since the data sources cannot be edited online, there is no need to call anything again, so all the code is inline here. Otherwise, it's the same general DWR call structure that by now you are quite familiar with. Here we encounter a new utility function that DWR offers, `dwr.util.removeAllOptions()`. This does what its name implies: it removes all options from a `<ul>`, `<ol>`, or `<select>` element. Here we're dealing with a `<select>` list. We first clear anything that might be in the list, and then proceed to add the new options, one for each data source.

The Exception Handler

The DWR exception handler that was set in `init()` is nothing but an `alert()` showing what was returned by the server, so I haven't shown it here.

Section Functions

We're now going to look at a group of functions that deals with the various functions that can be performed on a section, such as expanding and collapsing it. The first one we'll look at is the method that allows us to close a section entirely, namely `removeSection()`.

```
this.removeSection = function(inSectionName) {

  if (confirm("Are you sure you want to close that section?")) {
    new Effect.Fade(document.getElementById(inSectionName));
  }

}
```

After we confirm the user really wants to close the section, we use script.aculo.us to fade it out of view. Yep, it really is that simple!

Likewise, expanding a section that was previously collapsed couldn't be much simpler:

```
this.expandSection = function(inSectionName) {

  document.getElementById(inSectionName).style.height = "";
  new Effect.BlindDown(document.getElementById(inSectionName));

}
```

The `expandSection()` method takes care of it. Note that the height is first cleared, because we need the `BlindDown` effect to start with the section completely collapsed, not just collapsed to the header; otherwise, you see a nasty flicker.

Collapsing a section, accomplished with a call to the `collapseSection()` method, is a little bit more involved, as you can see:

```
this.collapseSection = function(inSectionName) {

  new Effect.BlindUp(document.getElementById(inSectionName), {
    afterFinish : function(inObj) {
     inObj.element.style.height = "40px";
     inObj.element.style.display = "block";
    },
    afterUpdate : function(inObj) {
      if (parseInt(inObj.element.style.height.replace("px", "")) < 40) {
        inObj.element.style.height = "40px";
      }
    }
  });

}
```

First, the `BlindUp` effect is used to "roll up" the section. However, two problems arise. First, we want the header portion of the section to still be visible. So, following each iteration of the effect loop, which is where the `afterUpdate` parameter comes into play, we check the height of the section. We do this by taking the reference to the object passed in, which is the section being collapsed, and we get its `style.height` attribute. Because this value is in the form `99px`, where `99` is the current height, we need to strip off the `px` portion, and then we compare the resultant integer (which we get with the `parseInt()` function), and if it's less than 40 pixels, we set the height of the section to 40 pixels. That effectively keeps the header portion always visible. One final step is required, and that's when the effect is all done, `afterFinish`, we make sure the final height of the section is 40 pixels and that it is fully displayed (because the effect itself will hide the section once it's completely rolled up).

The last method in this group of methods is `showHideAddSection()`, and this is the method that is responsible for showing (or hiding) one of the parts of a section seen when you click the various Add links (e.g., Add user, Add Report, etc.).

```
this.showHideAddSection = function(inDivID) {

  var section = document.getElementById(inDivID);
  if (section.style.display == "") {
    new Effect.Shrink(document.getElementById(inDivID));
  } else {
    new Effect.Grow(document.getElementById(inDivID));
  }

}
```

Here, we're doing a simple toggle based on the `display` property of the `<div>` the link was clicked for. The `Shrink` effect is shown when hiding the `<div>`, and `Grow` is used when showing it, which gives a nifty little animation to fight the boredom of what is actually being done!

Favorites Functions

The next group of methods we come across in `RePortalClass.js` are the functions dealing with favorites, beginning with the `updateFavoritesCallback()` that is called when the remote call to update the favorites list is complete:

```
this.updateFavoritesCallback = function(inList) {

  if (inList != null) {

    dwr.util.removeAllOptions("favorites_favoritesList");

    var formatter = function(inData) {
      var vals = inData.split("~~");
      return "<img src=\"img/icoDelete.gif\" align=\"absmiddle\" " +
        "style='cursor:pointer;'" +
        "onClick=\"RePortal.removeReportFromFavorites('" +
        vals[0] + "');\"" +
        "> <a href=\"javascript:void(0)\" " +
        "onClick=\"RePortal.runReport('" + vals[0] + "');\">" +
        vals[0] + "</a> - " + vals[1];
    };

    for (var i = 0; i < inList.length; i++) {
      dwr.util.addOptions("favorites_favoritesList",
        [ inList[i] ], formatter, { escapeHtml : false }
      );
    }

  }

}
```

This time, the usage of the `dwr.util.removeAllOptions()` function is working on a `<ul>` element, not a `<select>` element like we saw in `init()`, but it works just the same; we don't need to worry about the difference, DWR worries about it for us.

Next we find the definition of the `formatter`, which is a function. Let's skip that for just a moment though.

We then have a loop that iterates over the list of favorites returned by the call to the `FavoritesWorker.getFavoritesForUser()` remote method. For each, we use another new utility function: `dwr.util.addOptions`. It too does what its name implies: it adds an `<li>` or `<option>` to a `<ul>`/`<ol>` or `<select>` element. With each iteration of the loop and each call to `dwr.util.addOptions()`, that formatter now comes into play, as you can see it passed as the third parameter (the first parameter is the ID of the element the option is being added to, and the second is the data being used to create the option). The `formatter` accepts as its argument the data that is used to generate the option. In this case, it is a string in the form `xxx~~yyy`, where `xxx` is the name of the report and `yyy` is the description of it. The `formatter` function splits the string, and then creates a string of markup using the parts. The markup contains an

image that when clicked calls the `RePortal.removeReportFromFavorites()` method, passing in the name of the report (element zero of the array resulting from the `split()` call). It also includes a link that when clicked calls `RePortal.runReport()`, passing it the name of the report to run. Finally, the actual content of the option displayed is the name of the report and its description, the description being the second element in the array resulting from the call to `split()`.

You'll also notice that the call to `dwr.util.addOptions()` includes an object that contains options for it as the last parameter. The one option used here, `escapeHTML`, is needed because without it, the markup generated by the formatter will be displayed on the screen as-is, which obviously isn't what we want.

Adding a favorite is handled by the `addReportToFavorites()` method, and this simple method is seen here:

```
this.addReportToFavorites = function() {

  if (this.username == null) {
    alert("Sorry, you must be logged on to add favorites");
    return;
  }

  var reportName = dwr.util.getValue("favorites_reportsList");

  if (reportName == "") {
    alert("Sorry, you must select a report first");
    return;
  }

  reportName = reportName.split("~~")[0];
  FavoritesWorker.addReportToFavorites(this.username, reportName,
    { callback : RePortal.updateFavoritesCallback }
  );

}
```

After a check to ensure the user is logged in (otherwise he or she can't add favorites), the `dwr.util.getValue()` method is used to retrieve the name of the currently selected report, and this is checked to ensure a report is actually selected. Finally, a call is made to `FavoritesWorker.addReportToFavorites()`, which is the remote class on the server that handles the database update to add this favorite to the user's list of favorites. Finally, upon return from the server, the `updateFavoritesCallback()` method is called, which updates the favorites list in the UI so the new favorite shows up immediately.

Incidentally, in Figure 7-11 you can actually see what the UI looks like when adding a favorite.

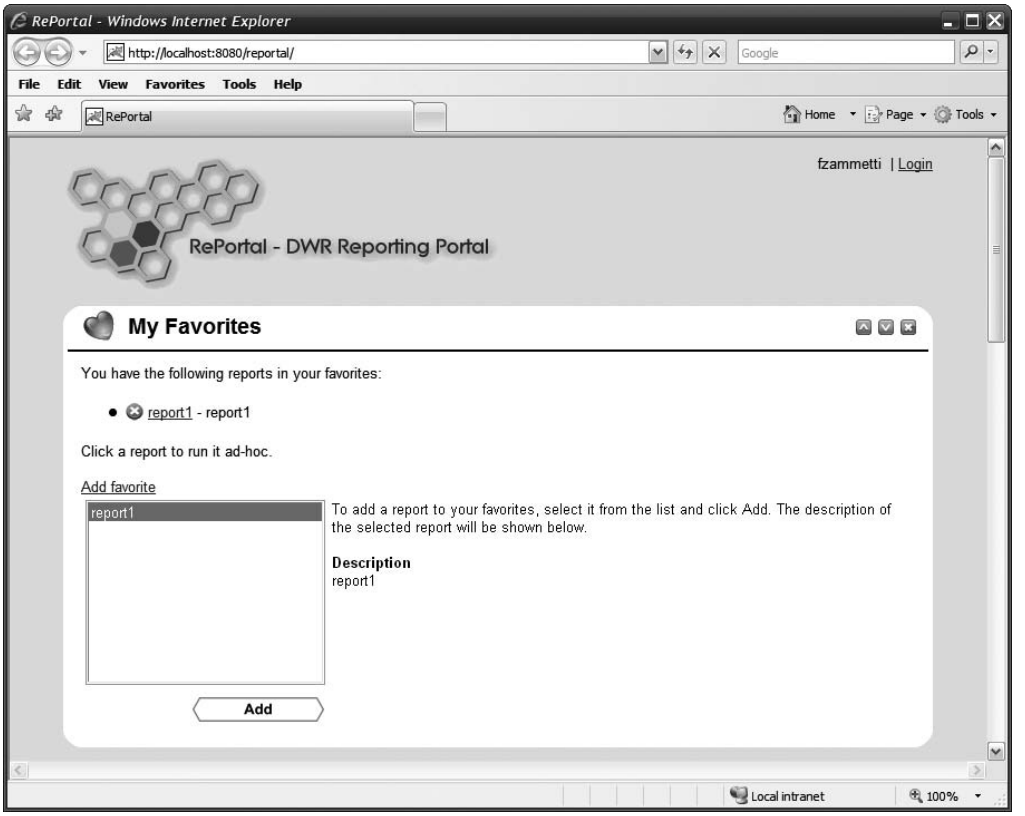

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
fzammetti | Login
RePortal - DWR Reporting Portal
My Favorites
You have the following reports in your favorites:
report1 - report1
Click a report to run it ad-hoc.
Add favorite
report1
To add a report to your favorites, select it from the list and click Add. The description of the selected report will be shown below.
Description
report1
Add
Local intranet
100%

Figure 7-11. *Adding a favorite*

One interesting thing is that when you select a report from the list, its description is shown to the right of the list, but it's done so with a little bit of flair, courtesy of another script.aculo.us effect. In web UI design, especially where Ajax is involved, the Yellow Fade Effect is often employed to get the user's attention pointed in the right direction. This effect is simple, yet effective: a piece of information that has just changed is highlighted in yellow and then fades to a nonhighlighted state. The code that accomplishes this is a call to `favoritesDisplayReportInfo()`:

```
this.favoritesDisplayReportInfo = function(inSelectValue) {

  var vals = inSelectValue.split("~~");
  document.getElementById("favorites_reportDescription").innerHTML = vals[1];
  new Effect.Highlight(
    document.getElementById("favorites_reportDescription"));

}
```

The term "Yellow Fade Effect" seems to have originated with a company called 37signals, as seen in this article: `www.37signals.com/svn/archives/000558.php`.

It's a simple matter of splitting the incoming value of the `<select>` (the second element of the resultant array is the description, because the value passed in will be in the form xxx~~yyy, as you may recall, where xxx is the report name and yyy is the description), displaying it in the appropriate `<div>`, and using the Highlight effect to do the yellow fade.

The final favorites-related function to examine is the `removeReportFromFavorites()` method, which is this:

```
this.removeReportFromFavorites = function(inReportName) {

  if (confirm("Are you sure you want to remove favorite for report '" +
    inReportName + "'?")) {
    FavoritesWorker.removeReportFromFavorites(this.username, inReportName,
      { callback : RePortal.updateFavoritesCallback }
    );
  }

}
```

As with most of these methods, there's not a whole lot to it, thanks to DWR. After a quick confirmation to be sure the user isn't destroying something he or she doesn't really want to, it's just a simple call to the `removeReportFromFavorites()` method of the remote `FavoritesWorker` class, passing in the username and report name, and the favorite is removed. Of course, the last step is to update the UI, just like when adding a favorite, and that's done the same way, with a call to `updateFavoritesCallback()`.

Scheduling Functions

The next group of functions to explore is those dealing with scheduling reports. The first is the method we saw called from `init()` that populates the list of scheduled reports, namely the `updateScheduledReportListCallback()` method.

```
this.updateScheduledReportListCallback = function(inList) {

  dwr.util.removeAllRows("reportScheduling_reportList");

  var cellFuncs = [
    function(data) {
      var vals = data.split("~~");
      return "<img src=\"img/icoDelete.gif\" align=\"absmiddle\" " +
      "style='cursor:pointer;'" +
      "onClick=\"RePortal.removeReportFromSchedule('" +
      vals[0] + "', '" + vals[1] + "', '" + vals[2] + "');\"" +
      "> " + vals[0];
    },
```

```
    function(data) { return data.split("~~")[1]; },
    function(data) {
      var runTime = data.split("~~")[2];
      var hour = runTime.split(":")[0];
      var minute = runTime.split(":")[1];
      runTime = "";
      if (parseInt(hour) < 10) {
        runTime += "0";
      }
      runTime += hour;
      runTime += ":";
      if (parseInt(minute) < 10) {
        runTime += "0";
      }
      runTime += minute;
      return runTime;
    },
    function(data) { return data.split("~~")[3]; },
    function(data) { return data.split("~~")[4]; }
  ];

  for (var i = 0; i < inList.length; i++) {
    var lastRun = inList[i].lastRun;
    if (lastRun == null) {
      lastRun = "None";
    } else {
      lastRun = "<a href=\"javascript:void(0);\" " +
      "onClick=\"RePortal.viewScheduledRun('" +
      inList[i].reportName + "', '" + inList[i].daysOfWeek + "', '" +
      inList[i].runTime.getHours() + ":" + inList[i].runTime.getMinutes() +
      "');\">" + inList[i].lastRunStatus + " - " + lastRun + "</a>";
    }
    dwr.util.addRows("reportScheduling_reportList",
      [ inList[i].reportName + "~~" + inList[i].daysOfWeek + "~~" +
        inList[i].runTime.getHours() + ":" + inList[i].runTime.getMinutes() +
        "~~" + inList[i].scheduledBy +
        "~~" + lastRun ], cellFuncs, { escapeHtml : false }
    );
  }

}
```

Ah, now this has some meat on its bone! To begin, we find a new function: `dwr.util.removeAllRows()`. This is akin to `dwr.util.removeAllOptions()`, which we saw earlier, but it works on tables. By using this, we clear the scheduled reports table, except for the header row, which will remain. (Go back and have a look at the markup in `index.jsp` for this table and notice that there is a defined `<thead>` and `<tbody>`. This function will only remove rows from the `<tbody>`.)

Next, we see something called `cellFuncs` defined. This is an array of functions, one for each column in a row of the table. When a row is later constructed, this array will be used, and the function for each column of the row being added will be called. This gives us the opportunity to generate the markup we need for each. As it happens, only the first and third columns really need any work, the other three simply output the value for the column. But before that makes sense, we need to examine the loop that follows the `cellFuncs` definition.

This loop iterates over the list that was returned from the server. Each element of the list is a JavaScript-equivalent instance of the server-side `ReportScheduleDescriptor` class. For each of these objects, we examine the `lastRun` field. If it's `null`, that means the scheduled report has not yet run, so the value "None" will be displayed. If it has run, we generate a link that allows the user to view that run of the report. To properly identify the data in the database to display, we need the name of the report, the days of the week it is scheduled to run, and the time it is schedule to run. These three pieces of information are essential for a composite key in the `schedules` table, so this information is pulled out of the `ReportScheduleDescriptor` object and passed to the `viewScheduledRun()` method as the `onClick` event handler of the link.

Next, for each `ReportScheduleDescriptor` object, a call to `dwr.util.addRows()` is sent, which, I'm sure you can guess, adds a row to the specified table. Now, for each of these calls, what happens is you can pass a string, which is the data for the row, and you can pass an array of functions that will be called for each column. That's where that `cellFuncs` array comes in. You'll note the string passed is constructed from the data in the `ReportScheduleDescriptor` object, and it's a double-tilde-separated sequence. Now if you look back at the functions in `cellFuncs`, it should make much more sense. For each column, we split the passed-in value and grab the appropriate data element. For three of the five columns, that value is simply returned, which is what the `dwr.util.addRows()` function will output for that column of the row.

For the other two, we do some processing to generate the appropriate markup. For the first column, that's the delete icon, plus the associated event handler to actually remove the report from the schedule (and again, the same three pieces of information are needed to uniquely identify the record in the table). The other column deals with the time the report is scheduled to run, and it's just code to format the time in a nice-looking way.

As we saw earlier with the `dwr.util.addOptions()` method, the `escapeHTML` option is required to ensure our markup is output as expected, and not as literal text.

Next up is `addReportToSchedule()`, which is a little lengthy, but not very complicated:

```
this.addReportToSchedule = function() {

  var hours = null;
  var minutes = null;
  hours = parseInt(dwr.util.getValue("reportScheduling_addHour"));
  minutes = parseInt(dwr.util.getValue("reportScheduling_addMinute"));
  if (hours < 0 || hours > 23 || minutes < 0 || minutes > 59) {
    alert("Sorry, the time you entered is not in a valid form");
    return;
  }

  var daysOfWeek = "";
  if (dwr.util.getValue("reportScheduling_addSunday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Sun";
```

```
  }
  if (dwr.util.getValue("reportScheduling_addMonday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Mon";
  }
  if (dwr.util.getValue("reportScheduling_addTuesday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Tue";
  }
  if (dwr.util.getValue("reportScheduling_addWednesday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Wed";
  }
  if (dwr.util.getValue("reportScheduling_addThursday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Thu";
  }
  if (dwr.util.getValue("reportScheduling_addFriday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Fri";
  }
  if (dwr.util.getValue("reportScheduling_addSaturday")) {
    if (daysOfWeek != "") { daysOfWeek += ","; } daysOfWeek += "Sat";
  }
  var runTimeDateObject = new Date();
  runTimeDateObject.setHours(hours);
  runTimeDateObject.setMinutes(minutes);
  runTimeDateObject.setSeconds(0);
  var reportScheduleDescriptor = {
    reportName :
      dwr.util.getValue("reportScheduling_addReportsList").split("~~")[0],
    daysOfWeek : daysOfWeek, scheduledBy : this.username,
    runTime : runTimeDateObject
  };

  if (reportScheduleDescriptor.daysOfWeek == "") {
    alert("Sorry, you must select at least one day of the week");
    return;
  } else if (reportScheduleDescriptor.reportName == "") {
    alert("Sorry, you must select a report to schedule");
    return;
  } else if (reportScheduleDescriptor.runTime == ":am" ||
    reportScheduleDescriptor.runtime == ":pm") {
    alert("Sorry, you must enter a time to run the report");
    return;
  }

  ReportSchedulingWorker.addReportToSchedule(reportScheduleDescriptor,
    { callback : RePortal.updateScheduledReportListCallback }
  );

}
```

First, a quick check is performed to ensure the time is entered in a valid 24-hour format. After that is an admittedly kind of ugly block of code whose job is simply to generate a string in the form `xxx,yyy,zzz`, where `xxx`, `yyy`, and `zzz` are three-letter abbreviations for the days of the week selected by the user (for instance, "Mon,Wed,Fri"). Following that, a `Date` object is instantiated, and the time is set to what was entered.

Now it's time to set up for the call, and to do this we need to construct a JavaScript equivalent of a server-side `ReportScheduleDescriptor` object. Of course, being JavaScript, that's as easy as creating a simple object with the appropriate fields present.

Once that object is ready, some validations are done to ensure the data is valid, and the method is aborted if anything isn't kosher. Finally, the call to the server is made, specifically to `ReportSchedulingWorker.addReportToSchedule()`, and as we've seen previously, the callback will update the UI to show the newly added report.

The UI as seen by the user when scheduling a new report is shown in Figure 7-12.

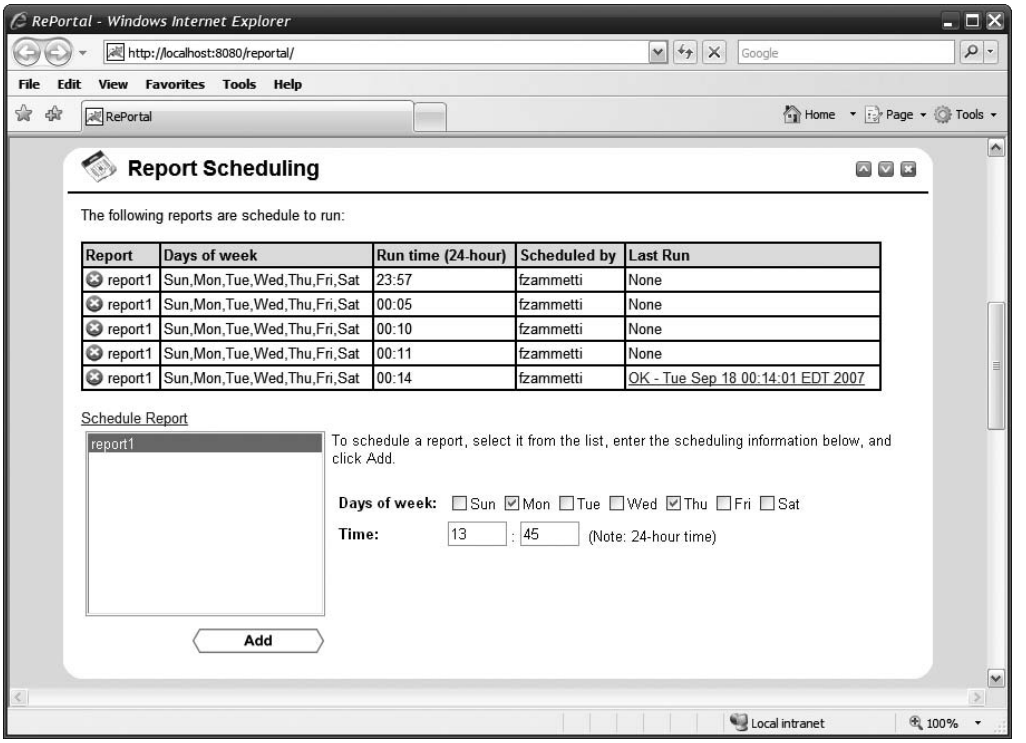

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
Google
File Edit View Favorites Tools Help
RePortal
Home Page Tools
Report Scheduling
The following reports are schedule to run:
Report | Days of week | Run time (24-hour) | Scheduled by | Last Run
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 23:57 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:05 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:10 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:11 | fzammetti | None
report1 | Sun,Mon,Tue,Wed,Thu,Fri,Sat | 00:14 | fzammetti | OK - Tue Sep 18 00:14:01 EDT 2007
Schedule Report
report1
To schedule a report, select it from the list, enter the scheduling information below, and click Add.
Days of week: Sun Mon Tue Wed Thu Fri Sat
Time: 13 : 45 (Note: 24-hour time)
Add
Local intranet
100%

Figure 7-12. *Scheduling a report*

Since we've seen adding a report, we should also see removing one:

```
this.removeReportFromSchedule = function(inReportName, inDaysOfWeek,
  inRunTime) {

  if (confirm("Are you sure you want to remove report '" +
    inReportName + "' from the schedule?")) {
    ReportSchedulingWorker.removeReportFromSchedule(inReportName,
```

```
      inDaysOfWeek, inRunTime,
      { callback : RePortal.updateScheduledReportListCallback }
    );
  }

}
```

This is very much along the lines of the method to remove a favorite that we examined earlier, so there's probably no need to rehash that here.

The final scheduling-related method is `viewScheduledRun()`, which is called when the user clicks the last column of the row for a report that has been run.

```
this.viewScheduledRun = function(inReportName, inDaysOfWeek,
  inRunTime) {

  lightboxPopup("divPleaseWait", true);
  ReportSchedulingWorker.viewScheduledRun(inReportName, inDaysOfWeek,
    inRunTime,
    {
      callback : function(inResp) {
        lightboxPopup("divPleaseWait", false);
        var reportWindow = window.open(null, inReportName,
          "width=780,height=550,scrollbars,resizable,toolbar");
        reportWindow.document.open();
        reportWindow.document.write(inResp);
        reportWindow.document.close();
      }
    }
  );

}
```

First, we see that the "Processing..." lightbox pop-up is shown. Next, the `ReportSchedulingWorker.viewScheduledRun()` method is called, passing it those three elements that make up the composite key that we've previously seen. The callback here then hides the "Processing..." pop-up and opens a new window. Finally, it writes what was returned by the server, which is the HTML generated when DataVision ran the report, to that new window. Nothin' to it!

Report Functions

At this point, you're going to notice that I start blasting through many of these methods. The reason is that like the markup in `index.jsp`, it kind of becomes a case of if you've seen one, you've seen them all. Most of the methods that are still left to look at are very similar to the ones we've seen already, so by and large I'll just point out important ways in which they differ and leave it at that.

That being said, we're now up to the group of functions dealing with report maintenance, and first up on the roll call is `updateReportListCallback()`, the method responsible for updating the lists (three in this case) of reports available on the portal.

```
this.updateReportListCallback = function(inList) {

  dwr.util.removeAllOptions("favorites_reportsList");
  dwr.util.removeAllOptions("reportScheduling_addReportsList");
  dwr.util.removeAllRows("reportMaintenance_reportList");

  var cellFuncs = [
    function(data) {
      return "<img src=\"img/icoDelete.gif\" align=\"absmiddle\" " +
      "style='cursor:pointer;'" +
      "onClick=\"RePortal.removeReportFromPortal('" +
      data.split("~~")[0] + "');\"" +
      "> " + data.split("~~")[0];
    },
    function(data) { return data.split("~~")[1]; },
    function(data) { return data.split("~~")[2]; }
  ];

  for (var i = 0; i < inList.length; i++) {
    dwr.util.addOptions("favorites_reportsList",
      [ { value : inList[i].reportName + "~~" + inList[i].description,
          text : inList[i].reportName } ],
      "value", "text"
    );
    dwr.util.addOptions("reportScheduling_addReportsList",
      [ { value : inList[i].reportName + "~~" + inList[i].description,
          text : inList[i].reportName } ],
      "value", "text"
    );
    dwr.util.addRows("reportMaintenance_reportList",
      [ inList[i].reportName + "~~" + inList[i].description + "~~" +
        inList[i].groups ], cellFuncs, { escapeHtml : false }
    );
  }

}
```

As you can see, we're dealing with three different lists of reports, one for the My Favorites section, one for the Report Scheduling section, and one for the Report Maintenance section. Aside from the fact that there are three of them updated by one method, and the fact that two of them are `<ul>`s and the other is a table, this is pretty much what you've seen before—no real surprises. But there is one new point, and that's in the calls to `dwr.util.addOptions()`. Note the `"value", "text"` arguments being passed. What these do is tell DWR the name of the attribute of the objects in the data array that are the value and text of the `<option>` element being added, respectively. This only applies when adding options to a `<select>`. To make this clearer: notice that the array being passed to the function is an array (a single-element array in this case, but an array nonetheless) of JavaScript objects. Further, note that the object has two attributes, `value` and `text`. So, by passing `"value", "text"`, we are telling the function that the

value attribute of the created <option> should come from the attribute named value of the object in the array, and the text the user sees on the screen should be taken from the text attribute. I could have just as easily named the attributes Archer and Tucker, and then I would have passed "Archer", "Tucker" as the arguments to the function, and it would have worked just the same.

Adding a report to the portal is accomplished by a call to the addReportToPortal() method, as seen here:

```
this.addReportToPortal = function() {

  var reportDescriptor = {
    reportName : dwr.util.getValue("reportMaintenance_addReportName"),
    description : dwr.util.getValue("reportMaintenance_addDescription"),
    groups : dwr.util.getValue("reportMaintenance_addGroups").toString(),
    dataSourceName :
      dwr.util.getValue("reportMaintenance_addDataSource").toString(),
    reportXML : dwr.util.getValue("reportMaintenance_addReportXML")
  };

  if (reportDescriptor.reportName == "") {
    alert("Sorry, you must supply a report name");
    return;
  } else if (reportDescriptor.groups == "") {
    alert("Sorry, you must select at least one group that can " +
      "access the report");
    return;
  } else if (reportDescriptor.reportXML == "") {
    alert("Sorry, you must supply the report XML");
    return;
  } else if (reportDescriptor.reportXML.length > 32000) {
    alert("Sorry, the report XML must be less than 32,000 characters long");
    return;
  } else if (reportDescriptor.dataSourceName == "") {
    alert("Sorry, you must select a data source");
    return;
  }

  ReportWorker.addReportToPortal(reportDescriptor,
    { callback : RePortal.updateReportListCallback }
  );

}
```

This is very much along the lines of addReportToSchedule(), so again, nothing too surprising. One thing to note is that toString() needs to be called on what dwr.util.getValue() returns when it is returning the value of multiple-enabled <select> elements; otherwise, the value returned is not in the correct form (a comma-separated list of the selected options).

We should probably get a quick look at the UI at this point, to see what adding a new report looks like, and Figure 7-13 is just such a look.

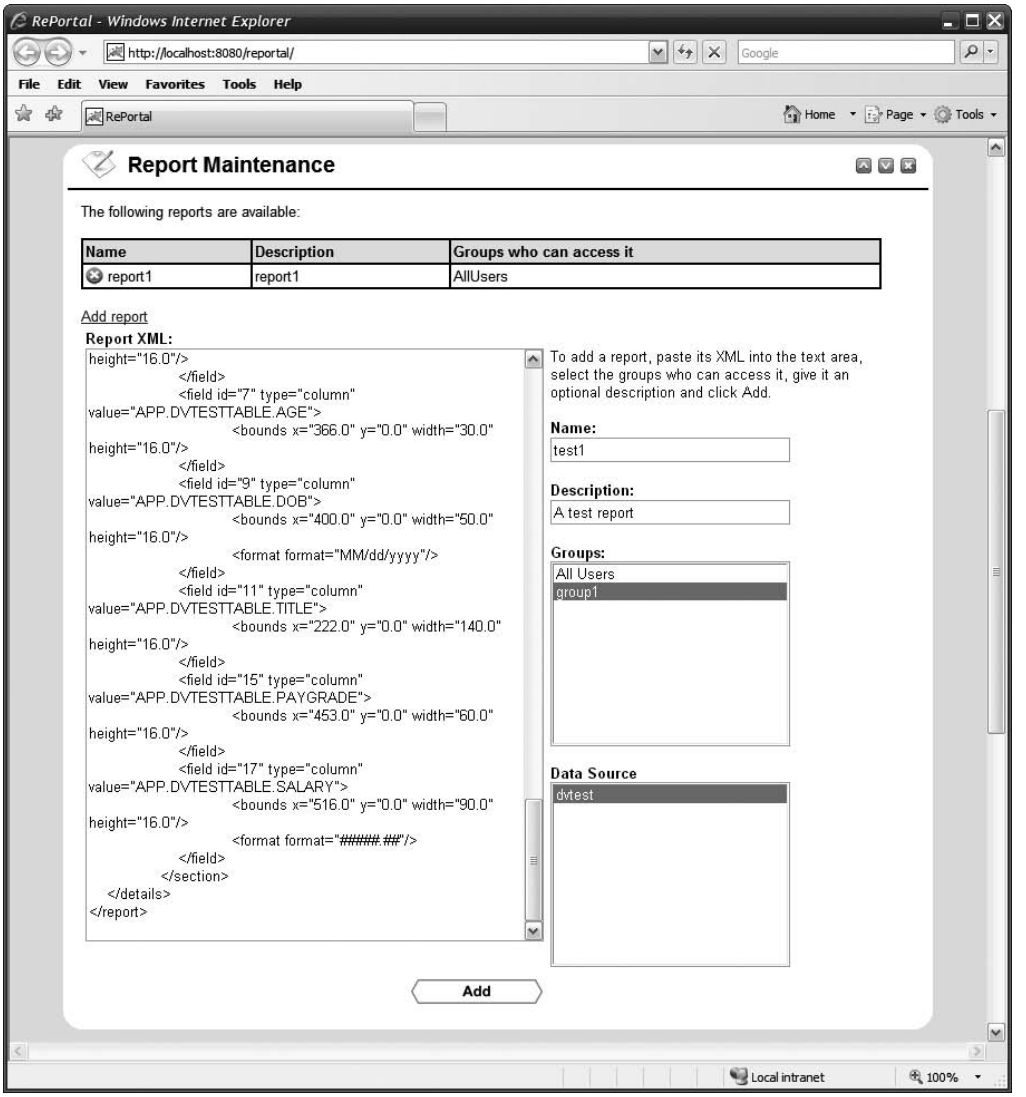


Figure 7-13. *Adding a report*

Removing a report is just as trivial as any of the other removal methods we've seen before, so I'm not even going to show it here. Take a peek for yourself though, you won't be surprised!

The final method in the group, `runReport()`, is responsible for . . . wait for it . . . **running a report!**

```
this.runReport = function(inReportName) {

  lightboxPopup("divPleaseWait", true);
  ReportRunner.runReport(inReportName,
    {
      callback : function(inResp) {
        lightboxPopup("divPleaseWait", false);
        var reportWindow = window.open(null, inReportName,
          "width=780,height=550,scrollbars,resizable,toolbar");
        reportWindow.document.open();
        reportWindow.document.write(inResp);
        reportWindow.document.close();
      }
    }
  );

}
```

As you can see, it's very simple: show the "Processing..." lightbox pop-up, and then call `ReportRunner.runReport()`, passing it the name of the report to run. Upon successful return from the server, the lightbox is hidden, a new window is opened, and the HTML generated by DataVision is written to it.

Group Functions

Now we come to the group of functions dealing with groups, beginning with the method to update the list of groups, `updateGroupsListCallback()`.

```
this.updateGroupsListCallback = function(inList) {

  dwr.util.removeAllOptions("reportMaintenance_addGroups");
  dwr.util.removeAllOptions("groupAdministration_groupsList");
  dwr.util.removeAllOptions("userAdministration_addGroups");
  dwr.util.addOptions("reportMaintenance_addGroups",
    [ { value : "AllUsers", text : "All Users" } ], "value", "text"
  );
  for (var i = 0; i < inList.length; i++) {
    dwr.util.addOptions("reportMaintenance_addGroups",
      [ { value : inList[i].groupName, text : inList[i].groupName } ],
      "value", "text"
    );
    dwr.util.addOptions("userAdministration_addGroups",
      [ { value : inList[i].groupName, text : inList[i].groupName } ],
      "value", "text"
    );
    dwr.util.addOptions("groupAdministration_groupsList",
      [ { value : inList[i].groupName + "~~" + inList[i].description,
          text : inList[i].groupName } ],
```

```
      "value", "text"
    );
  }

}
```

The only real difference between this and the other similar methods for favorites and reports is that there is an option that is always available, the All Users pseudo-group, and you can see that it's added manually before the iteration over whatever groups may have been returned from the server is done.

To add a group to the portal, the following method is executed:

```
this.addGroupToPortal = function() {

  var groupDescriptor = {
    groupName : dwr.util.getValue("groupAdministration_addGroupName"),
    description : dwr.util.getValue("groupAdministration_addDescription")
  };

  if (groupDescriptor.groupName == "") {
    alert("Sorry, you must supply a group name");
    return;
  }

  GroupWorker.addGroupToPortal(groupDescriptor,
    { callback : RePortal.updateGroupsListCallback }
  );

}
```

No surprises there, or so I'd hope at this point! The screen looks like Figure 7-14 when adding a group.

Removing a group is likewise exactly what you'd expect:

```
this.removeGroupFromPortal = function() {

  var groupName = dwr.util.getText("groupAdministration_groupsList");
  if (groupName) {
    if (confirm("Are you sure you want to remove group '" +
      groupName + "'?")) {
      dwr.util.setValue("groupAdministration_groupDescription", "");
      GroupWorker.removeGroupFromPortal(groupName,
        { callback : RePortal.updateGroupsListCallback }
      );
    }
  }

}
```

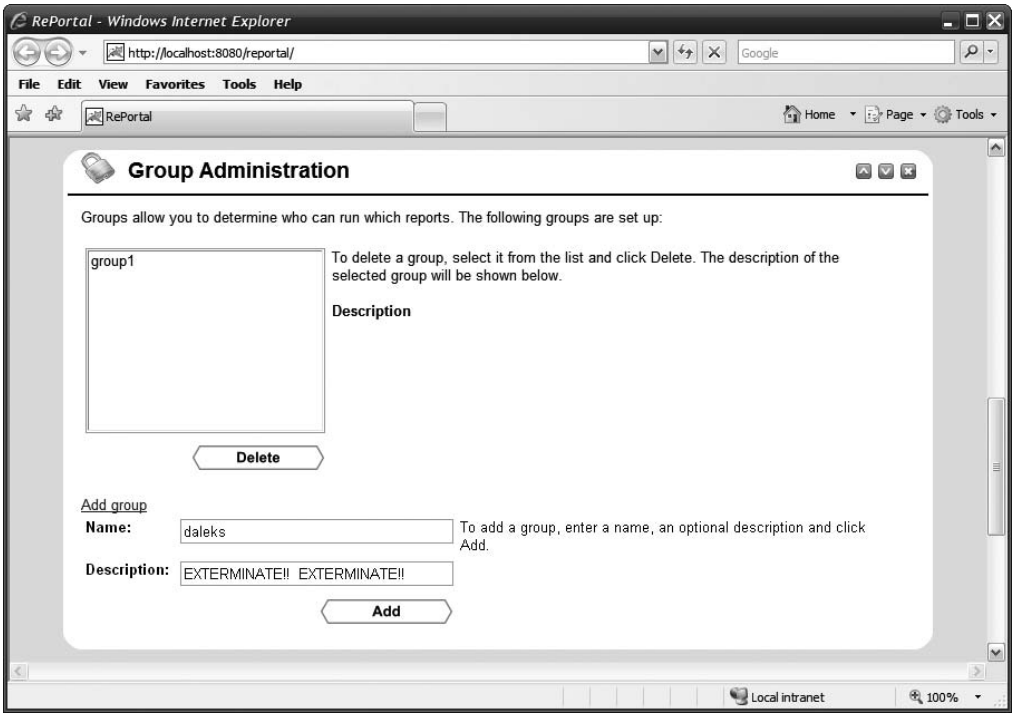

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
Group Administration
Groups allow you to determine who can run which reports. The following groups are set up:
group1
To delete a group, select it from the list and click Delete. The description of the selected group will be shown below.
Description
Delete
Add group
Name:
daleks
To add a group, enter a name, an optional description and click Add.
Description:
EXTERMINATE!! EXTERMINATE!!
Add
Local intranet
100%

Figure 7-14. *Adding a group*

Notice the `dwr.util.getText()` function being used rather than `dwr.util.getValue()`. The `dwr.util.getText()` function, which we haven't seen before, gets the text of an `<option>` element for a given `<select>`, rather than its value. This would have worked either way, since the value and the text happen to be the same, but this gave me an opportunity to introduce this new function. Obviously, sometimes you'll want the value vs. the text, so it's good to know you can do either with DWR easily.

The final method, `displayGroupInfo()`, I've chosen not to show here as it's very small and identical to the `displayReportInfo()` method that we saw earlier, so I figured I may as well save some space in an already very long chapter!

User Functions

Also in an attempt to shave a few pages off the total page count, most of the user-related functions aren't shown here, because frankly if you went through the group functions, you can basically replace the word "group" in them with "user," and you have practically seen the code anyway!

I will, however, take this opportunity to show you the UI when adding a user, which you can see in Figure 7-15.

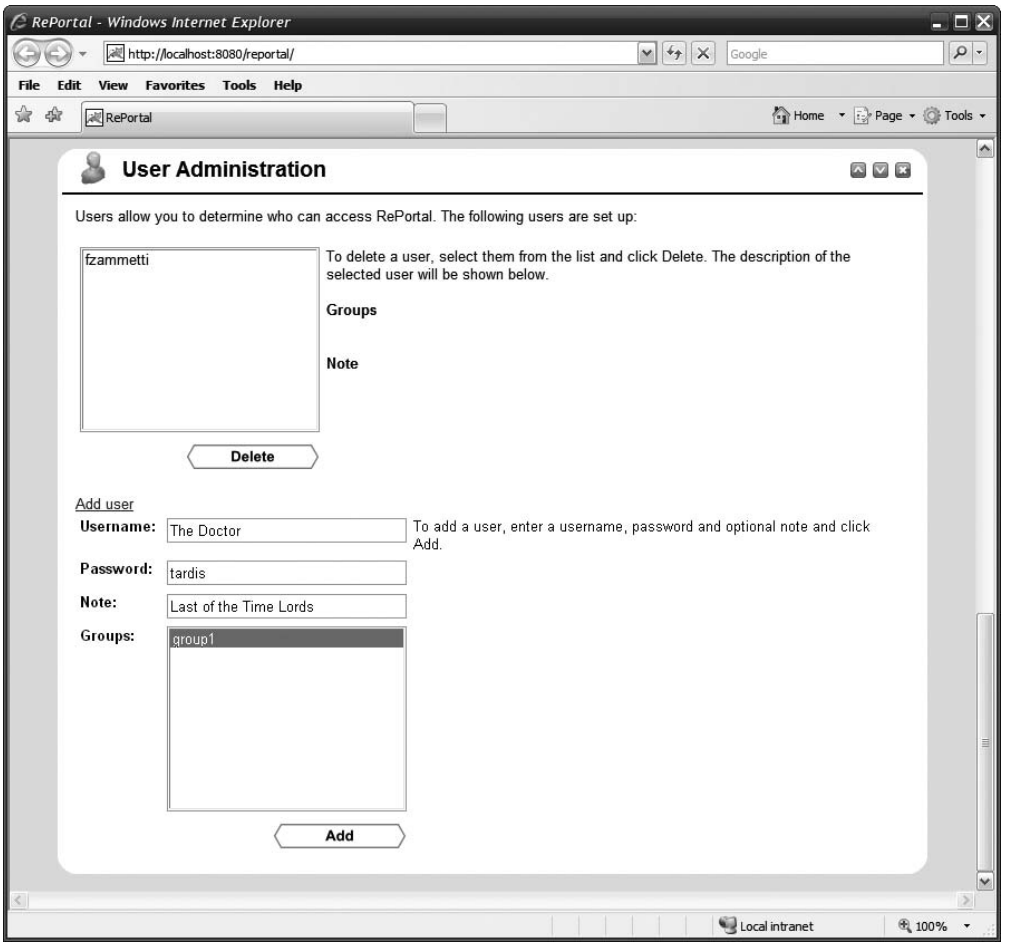

RePortal - Windows Internet Explorer
http://localhost:8080/reportal/
File Edit View Favorites Tools Help
RePortal
Home Page Tools
User Administration
Users allow you to determine who can access RePortal. The following users are set up:
fzammetti
To delete a user, select them from the list and click Delete. The description of the selected user will be shown below.
Groups
Note
Delete
Add user
Username: The Doctor
To add a user, enter a username, password and optional note and click Add.
Password: tardis
Note: Last of the Time Lords
Groups: group1
Add
Local intranet
100%

Figure 7-15. *Adding a user*

I'll also point out the `displayUserInfo()`, which differs slightly from `displayGroupInfo()` because it highlights two values at once, as you can see:

```
this.displayUserInfo = function(inSelectValue) {

  var vals = inSelectValue.split("~~");
  dwr.util.setValue("userAdministration_userGroups", vals[0]);
  dwr.util.setValue("userAdministration_userNote", vals[1]);
  new Effect.Highlight(dwr.util.byId("userAdministration_userGroups"));
  new Effect.Highlight(dwr.util.byId("userAdministration_userNote"));

}
```

Lastly, the `logUserIn()` method is unique to users:

```
this.logUserIn = function() {

  UserWorker.logUserIn(dwr.util.getValue("login_username"),
    dwr.util.getValue("login_password"),
    function(inResp) {
      if (inResp) {
        lightboxPopup("divLogin", false);
        RePortal.init(dwr.util.getValue("login_username"));
      } else {
        alert("Sorry, information was invalid, could not log you in");
        lightboxPopup("divLogin", false);
      }
    }
  );

}
```

Sure, understanding it isn't exactly like trying to pass the Star Fleet entrance exam (What's the correct matter-to-antimatter ratio for a warp core?[9]), but what the heck, let's check it out anyway. This is called when the user clicks the login button on the login lightbox pop-up. It's a simple matter of a call to `UserWorker.logUserIn()`, passing it the username and password the user entered. If null is returned, the user isn't logged in, which is the `else` portion of the branch. When the user is logged in, however, the `init()` method is called again to update the UI and show all the other sections and all the other data. It's simple, but it gets the job done.

The Server-Side Code

And now, after our long Trek (heh heh) across the frozen tundra of Andoria . . . err, the client side of RePortal, it's finally time to look at the server side of things. We begin our journey of discovery with a quick look at a very simple class, `Config`.

Config.java

The `Config` class stores, via static members, the data read in from the `appConfig.xml` file. The UML diagram for this class is shown in Figure 7-16.

---

9. In the *Next Generation* episode "Coming of Age," we get to see the boy wonder himself, Wesley Crusher, taking the exam. The question about the matter-to-antimatter ratio is a trick question asked on the exam. The correct answer, of course, is 1:1, and in fact no other ratio makes sense in such a reaction (hence the reason it's a trick question) because if either matter or antimatter is present in greater quantity than the other, there will always be some matter that isn't annihilated, releasing energy. The 1:1 ratio is the most optimal since all reactant is converted to energy with no waste (and just to prove how big a nerd I actually am, I actually knew that answer as I watched the episode the first time before they stated it . . . any self-respecting science geek would!).

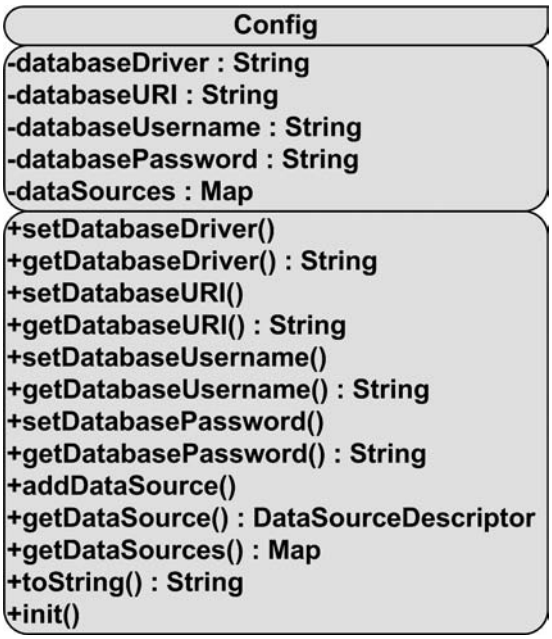


Figure 7-16. *UML diagram of the* `Config` *class*

This class contains almost nothing but a bunch of getters and setters, and they aren't shown here lest I really put you to sleep. However, there's also an `init()` method that is rather more interesting:

```
public void init(final InputStream inConfigFileStream) throws IOException,
  SAXException {

  Digester digester = new Digester();
  digester.setValidating(false);

  digester.addObjectCreate("appConfig",
    "com.apress.dwrprojects.reportal.Config");
  digester.addBeanPropertySetter("appConfig/databaseDriver",
    "databaseDriver");
  digester.addBeanPropertySetter("appConfig/databaseURI", "databaseURI");
  digester.addBeanPropertySetter("appConfig/databaseUsername",
    "databaseUsername");
  digester.addBeanPropertySetter("appConfig/databasePassword",
    "databasePassword");
  digester.addObjectCreate("appConfig/dataSources/dataSource",
    "com.apress.dwrprojects.reportal.DataSourceDescriptor");
  digester.addBeanPropertySetter("appConfig/dataSources/dataSource/name",
    "name");
  digester.addBeanPropertySetter(
    "appConfig/dataSources/dataSource/description", "description");
```

```
    digester.addBeanPropertySetter(
      "appConfig/dataSources/dataSource/databaseDriver",
      "databaseDriver");
    digester.addBeanPropertySetter(
      "appConfig/dataSources/dataSource/databaseURI", "databaseURI");
    digester.addBeanPropertySetter(
      "appConfig/dataSources/dataSource/databaseUsername",
      "databaseUsername");
    digester.addBeanPropertySetter(
      "appConfig/dataSources/dataSource/databasePassword",
      "databasePassword");
    digester.addSetNext("appConfig/dataSources/dataSource",
      "addDataSource",
      "com.apress.dwrprojects.reportal.DataSourceDescriptor");

    digester.parse(inConfigFileStream);

}
```

This is the code responsible for getting `appConfig.xml` into the `Config` class. It uses a library called Commons Digester, from the Jakarta branch of Apache. If you've never seen it before, it's quite a fantastic creation. Let's break this code down . . .

First, we have two lines of code to set things up. The first line, obviously, instantiates a `Digester` object. You can reuse this object as required, provided any previous parse activities have been completed and you do not try to use the same instance from two different threads. The second line of code tells `Digester` that we do not want the XML document validated against a DTD.

After that comes a series of `addXXX` method calls, which each adds a particular rule to `Digester`. A number of built-in rules are available, and you can write your own as required.

All of the rules share the first method call parameter in common: the path to the element the rule will fire for. Recall that an XML document is a hierarchical tree structure, so to get to any particular element in the document, you form a "path" to it that starts at the document root and proceeds through all the ancestors of the element. In other words, looking at the `<databaseDriver>` elements, the parent of that is the `<appConfig>` element. Therefore, the full path to the `<databaseDriver>` element is `appConfig/databaseDriver`. Likewise, for the `<databaseDriver>` that has a `<dataSource>` as its parent, the path is `appConfig/dataSource/databaseDriver`. A `Digester` rule is "attached" to a given path and will fire any time an element with that path is encountered. You can have multiple rules attached to a given path, and multiple rules can fire for any given path. It should be noted that some rules fire when an element begins, others when it ends, and still others both times.

In this example, our first rule, an `ObjectCreate` rule, is defined to fire for the path `appConfig`. This means that when the `<appConfig>` element is encountered, an instance of the class `com.apress.dwrprojects.reportal.Config` will be created.

`Digester` uses a stack implementation to deal with the objects it creates. For instance, when the `ObjectCreate` rule fires and instantiates that `Config` object, it is pushed onto the stack. All subsequent rules, until the object is popped off the stack either explicitly as a result of another rule or because parsing is completed, will work against that object. So, when the next rule, the `addBeanPropertySetter` rule, fires, it will set the given property (`databaseDriver`)

of the object on the top of the stack, in this case our `Config` object. There are a series of `addBeanPropertySetter` rules, one for each of the children directly under the `<appConfig>` element.

Then there is another `ObjectCreate` rule set up for the `<dataSource>` elements that creates a `DataSourceDescriptor` object. So, when the first `<dataSource>` element is encountered, the object is created and pushed onto the stack, meaning it is now on top of the `Config` object. There is a batch of `addBeanPropertySetter` rules for the children of the `<dataSource>` element, and they will be setting properties on the `DataSourceDescriptor` object on the top of the stack.

Lastly, we see a `SetNext` rule. What this does is call a given method, `addDataSource()` in this case, on the **next** object on the stack, which would be the `Config` object, remember, passing it the object on the top of the stack, the `DataSourceDescriptor` object. At the end of this, the `DataSourceDescriptor` object on the top of the stack is popped off, revealing the `Config` object, which is again the top object on the stack. This process repeats until all the data in the XML file has been processed.

At the end, the object on the top of the stack, which would be our `Config` object at that point, is popped and returned by Digester. You could grab that return if you needed to do something with the object, but in this case there's no need to.

I hope you agree that Digester makes parsing XML into objects incredibly easy. For my money, it's one of the most useful Commons components available.

COMMONS DIGESTER: XML PARSING WITHOUT (AS MUCH) PAIN

If you've ever written SAX code, which Digester uses under the covers, you'll really appreciate what Digester is doing for you. Parsing XML is one of those things that they say any third-year computer science student should be able to do with ease, but in the real world it tends to not be quite as simple as that. Digester takes (most) of the pain out of it.

As I mentioned, there are a bunch of rules that Digester offers, and I encourage you to take a peek at all it has to offer here: `http://commons.apache.org/digester`. It's not the only game in town, to be sure, but it's one of the best in this author's opinion.

ContextListener.java

The `ContextListener` class is your basic, standard servlet context listener class, so there's no UML diagram here. Further, I haven't shown the entire code listing here, just the interesting part, the `contextInitialized()` method.

```
public void contextInitialized (
  final ServletContextEvent inServletContextEvent) {

  try {

    ServletContext servletContext = inServletContextEvent.getServletContext();
    InputStream isConfigFile =
      servletContext.getResourceAsStream(CONFIG_FILE);
```

```
    try {
      Config config = new Config();
      config.init(isConfigFile);
      log.info("contextInitialized() - Config = " + config);
    } catch (IOException ioe) {
      ioe.printStackTrace();
      log.error("contextInitialized() - Unable to read app config file.  " +
      "App *NOT* initialized!");
    } catch (SAXException se) {
      se.printStackTrace();
      log.error("contextInitialized() - Unable to parse app config file.  " +
        "App *NOT* initialized!");
    }

    DatabaseWorker databaseWorker = new DatabaseWorker();
    databaseWorker.validateDatabase();
    ReportSchedulingWorker rsw = new ReportSchedulingWorker();
    rsw.setDatabaseWorker(databaseWorker);
    rsw.startScheduler();

  } catch (Exception e) {
    log.error("contextInitialized() - " +
      "Exception occurred during RePortal initialization.  " +
      "Application WILL NOT be available.  Exception was: " + e);
  }

}
```

We begin by getting a stream to the `appConfig.xml` file. Once we have that, we pass that along to the `init()` method of the `Config` class, which we just examined a short time ago. Assuming no exceptions are thrown by that process, we go on to instantiate a `DatabaseWorker` and call the `validateDatabase()` method. This is all virtually identical to what we already saw in the DWiki application.

However, we then come to something a bit new, and that's the `ReportSchedulingWorker` stuff. An instance of that class is created, and the `DatabaseWorker` instance is handed to it. Then, we call `startScheduler()`. This will kick off the Quartz scheduler so all scheduled report jobs can run.

DatabaseWorker.java

As stated previously, the `DatabaseWorker` class is basically the same as what we saw in DWiki. The only real differences are in the SQL statements used to validate and create the database tables. Since we've already looked at the database schema, the actual SQL shouldn't be any surprise at all, so have a quick look at this class to convince yourself there's nothing new and exciting going on and that the SQL matches your expectations based on the schema, and let's move along.

DataSourceDescriptor.java

The `DataSourceDescriptor` class is a simple VO that stores information about a given data source configured in the `appConfig.xml` file. Its UML diagram can be seen here in Figure 7-17.

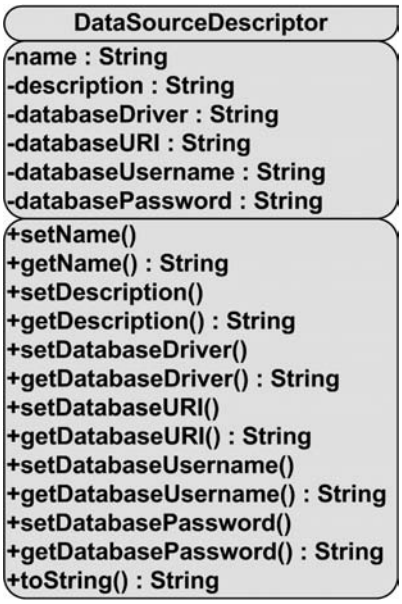


Figure 7-17. *UML diagram of the* `DataSourceDescriptor` *class*

No code is shown here as it's just a collection of private fields with public getters and setters for each.

FavoritesWorker.java

The `FavoritesWorker` is the first of the `xxxWorker` classes, not counting `DatabaseWorker`, which is named similarly but is really a different beast. These are the beans exposed to DWR for remoting, and they form the core code of RePortal on the server. In Figure 7-18, you can see the UML diagram for `FavoritesWorker`.

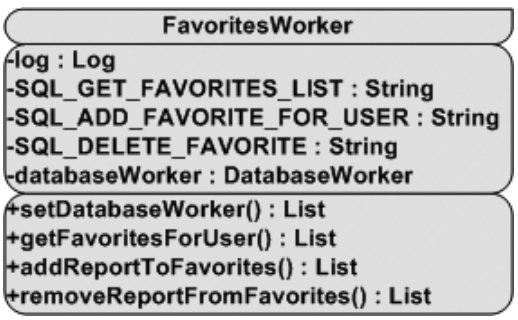


Figure 7-18. *UML diagram of the* `FavoritesWorker` *class*

First, let me go over some generic things about these xxxWorker classes that you'll see repeatedly, but which I won't talk about repeatedly. First, each one contains a number of static final strings that are the SQL queries they need to do their jobs. For each class, I'll show a table, like Table 7-4 here, which tells you the query itself, the name of the field in the class, and a brief description of each. The queries themselves are very simple and shouldn't give you much trouble.

Table 7-4. *The SQL Statements Contained in the* FavoritesWorker *Class*

SQL Statement	Class Field	Description
SELECT f.reportname, r.description, r.groups FROM favorites f, reports r WHERE (f.username='${username}' OR r.groups LIKE ' %AllUsers%') AND f.reportname=r.reportname	SQL_GET_FAVORITES_LIST	Returns the list of favorites for a given user and/or reports made available to all users
INSERT INTO favorites (username, reportname) values('${username}', '${reportname}')	SQL_ADD_FAVORITE_FOR_USER	Adds a favorite for a given user
DELETE FROM favorites WHERE username='${username}' and reportname='${reportname}'	SQL_DELETE_FAVORITE	Removes a favorite from a user's list of favorites

One thing you'll see is that most of the statements include dynamic tokens in the form ${xxx}. These are replaced with the actual values by the DatabaseWorker class, as you saw in the DWiki project, but just keep that in mind as you review the SQL so you aren't scratching your head wondering what that's all about.

The next generic bit of code you'll see is this:

```
  private DatabaseWorker databaseWorker;

  public void setDatabaseWorker(final DatabaseWorker inDatabaseWorker) {

    databaseWorker = inDatabaseWorker;

  }
```

Every xxxWorker class has this, and you may recall from our look at the spring-beans.xml file that a DatabaseWorker instance will be injected into the class when instantiated by DWR.

That takes care of the generic stuff. Now we can move on to the code specific to the FavoritesWorker, beginning with the getFavoritesForUser() method.

```
@SuppressWarnings("unchecked")
public List<String> getFavoritesForUser(final String inUsername)
  throws Exception {

  try {
```

```
    Map<String, String> tokens = new HashMap<String, String>();
    if (inUsername == null) {
      tokens.put("username", "_DUMMY_USERNAME_");
    } else {
      tokens.put("username", inUsername);
    }

    List<Map> favorites =
      databaseWorker.executeQuery(SQL_GET_FAVORITES_LIST, tokens);

    List<String> favoritesList = new ArrayList<String>();
    for (Map m : favorites) {
      favoritesList.add((String)m.get("REPORTNAME") + "~~" +
        (String)m.get("DESCRIPTION"));
    }
    return favoritesList;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

First, note the `@SuppressWarnings` annotation. This is used to turn off some warnings that will be generated during the compile phase because of the call to `databaseWorker.executeQuery()`. It actually traces all the way back to the Spring `JdbcTemplate`'s `queryForList()` call, which doesn't return a type-specific collection. So, instead of seeing the warnings at compile time, since I know it's safe to do so, this annotation suppresses them. You'll see this on a number of methods in these classes, but they all serve the same basic purpose, owing to the same underlying cause, so consider this part of that generic stuff I won't be mentioning again!

Next we see a `Map` of tokens being created that will be used to replace the dynamic tokens in the SQL query. Included in this `Map` is a username element. Now, when this method is called when a user isn't logged in, no username is passed. In that case, a dummy value of `_DUMMY_USERNAME_` is used. This is done so that the query won't be broken by a null username. Unless someone explicitly adds a user with that username, the result will be that the query won't find any matches based on username, but it **will** still find matches for all users, so things work as expected this way.

Finally, the query is executed and the returned `List` is iterated over. For each item returned, we construct a string in the form `xxx~~yyy`, where `xxx` is the name of the report and `yyy` is the description, and we add that string to a `List`, which is returned. We previously saw how the UI uses this string, so now the picture is complete.

One last bit of generic code is the exception handling. For all the `xxxWorker` classes, a generic `Exception` is caught and its stack trace printed to stdout for debugging, and it is rethrown. The UI will handle it via the exception handler we set up with DWR.

Next up is the addReportToFavorites() method:

```
public List<String> addReportToFavorites(final String inUsername,
  final String inReportName) throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("username", inUsername);
    tokens.put("reportname", inReportName);

    databaseWorker.executeUpdate(SQL_ADD_FAVORITE_FOR_USER, tokens);
    return getFavoritesForUser(inUsername);

  } catch (DataIntegrityViolationException dive) {
    throw new Exception("Favorite could not be created.\n\n" +
      "(Does the favorite already exist for the specified report?)");
  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

As with getFavoritesForUser(), it's the same basic pattern: populate a Map to use to replace tokens in the SQL query, use databaseWorker to execute the query. But, now there's something interesting that you're also going to see quite a bit of, and that's the call to getFavoritesForUser() and its result being returned from addReportToFavorites(). For most of the methods we'll look at, there is an associated UI update. Here, we need to update the list of favorites on the screen. I could have made a separate call, triggered from the callback to the DWR call that executed this method, to retrieve the updated list of favorites, and then update the screen. This is rather inefficient though—since we know we need to update the screen following this call, why not just go ahead and return the data we'll need from this method and save one round-trip? That's exactly what calling getFavoritesForUser() does for us here.

Much like in the RePortal.js code, you're going to find a fairly repetitive pattern emerging here, so I'll gloss over some things as we go as that repetitiveness starts to come out; but for this class, since it's the first xxxWorker class, we'll look at all of its code.

Next up is `removeReportFromFavorites()`:

```
public List<String> removeReportFromFavorites(final String inUsername,
  final String inReportName) throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("username", inUsername);
    tokens.put("reportname", inReportName);

    databaseWorker.executeUpdate(SQL_DELETE_FAVORITE, tokens);
    return getFavoritesForUser(inUsername);

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Once again, we find the same basic structure as the last two methods, and the same retrieving of the updated data to be returned to the client as in the last method.

GroupDescriptor.java

The `GroupDescriptor` class is a simple VO that stores information about a given group that has been created in the portal. Its UML diagram can be seen here in Figure 7-19.

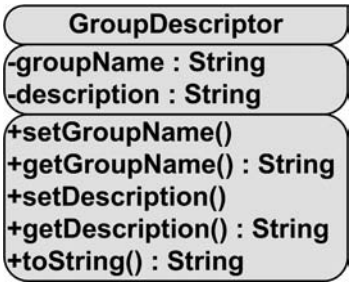


Figure 7-19. *UML diagram of the* `GroupDescriptor` *class*

No code is shown here as it's just a collection of private fields with public getters and setters for each.

GroupWorker.java

The `GroupWorker` class deals with the operations pertaining to groups. You can see the UML diagram of this class in Figure 7-20.

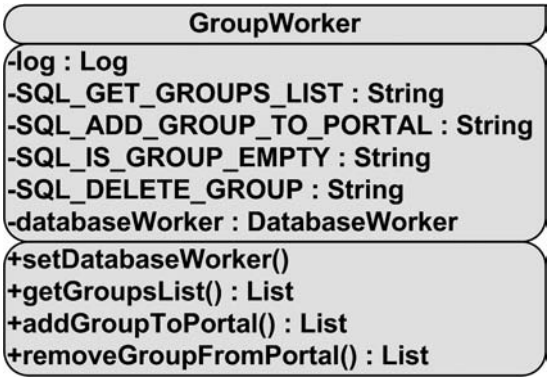


Figure 7-20. *UML diagram of the* `GroupWorker` *class*

As before, there are a couple of SQL queries present in this class to allow it to do its thing, and those queries are summarized in Table 7-5.

Table 7-5. *The SQL Statements Contained in the* `GroupWorker` *Class*

SQL Statement	Class Field	Description
`SELECT * FROM groups`	`SQL_GET_GROUPS_LIST`	Returns a list of all groups known to the portal
`INSERT INTO groups (groupname, description) VALUES('${groupname}', '${description}')`	`SQL_ADD_GROUP_TO_PORTAL`	Adds a group to the portal
`SELECT groups FROM users WHERE groups LIKE '%${groupname}%'`	`SQL_IS_GROUP_EMPTY`	Checks to see whether any users are assigned to a given group (which is needed before a group can be deleted)
`DELETE FROM groups WHERE groupname='${groupname}'`	`SQL_DELETE_GROUP`	Deletes a given group

The first method we stumble upon, like a drunken yeoman who dipped too far into a bottle of Romulan Ale, is the `getGroupLists()` method:

```
@SuppressWarnings("unchecked")
public List<GroupDescriptor> getGroupsList() throws Exception {

  try {

    List<Map> groups = databaseWorker.executeQuery(SQL_GET_GROUPS_LIST,
      new HashMap());
    List<GroupDescriptor> groupList = new ArrayList<GroupDescriptor>();

    for (Map m : groups) {
      GroupDescriptor group = new GroupDescriptor();
      group.setGroupName((String)m.get("GROUPNAME"));
```

```
      group.setDescription((String)m.get("DESCRIPTION"));
      groupList.add(group);
    }
    return groupList;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

This is again pretty typical code like we've seen before, but this time you'll note we're returning a `List` not of strings, but of a specified data type: `GroupDescriptor`. The callback on the client uses these objects to populate the list of groups. You may have cause to wonder why I didn't have a `FavoritesDescriptor` and return a `List` of those from the `getFavoritesForUser()` method in the `FavoritesWorker` class. The answer is simply that it's a good way to show a different approach to the same basic need, so for demonstration purposes it was useful. There's probably no **technical** reason this method and the other couldn't have both been done the same way, whichever way you prefer. But hey, this is a book, you're supposed to be exposed to new ideas, and like Seven of Nine, you can never have too many curves (and if you're a lady reading this, you can think of the curves on Jean-Luc Picard's bald head, I suppose).

Next up is the `addGroupToPortal()` method, shown here:

```
public List<GroupDescriptor> addGroupToPortal(final GroupDescriptor inGroup)
  throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("groupname", inGroup.getGroupName());
    tokens.put("description", inGroup.getDescription());

    databaseWorker.executeUpdate(SQL_ADD_GROUP_TO_PORTAL, tokens);
    return getGroupsList();

  } catch (DataIntegrityViolationException dive) {
    throw new Exception("Group could not be created.  " +
      "Does the group already exist?");
  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Oh look! Something new! The `catch` of the `DataIntegrityVoilationException`, which is a Spring abstraction around a `SQLException`, is there in case a group is added that already exists (based on `groupName`). I didn't see any way to determine the exact cause of the failure (maybe someone can e-mail me what I missed), so that's why the message returned is framed as a question (although this should be the only possible cause anyway).

The `removeGroupFromPortal()` method follows that, and here's that code, lest I keep you in suspense any longer:

```
@SuppressWarnings("unchecked")
public List<GroupDescriptor> removeGroupFromPortal(final String inGroupName)
  throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("groupname", inGroupName);

    List<Map> groupRecords = databaseWorker.executeQuery(SQL_IS_GROUP_EMPTY,
      tokens);
    if (groupRecords.size() > 0) {
      throw new Exception("Group cannot be deleted because there are users " +
        "assigned to it.  Delete all users in the group first.");
    }

    databaseWorker.executeUpdate(SQL_DELETE_GROUP, tokens);
    return getGroupsList();

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Here too we have something different! Before a group can be deleted, we have to ensure there are no users assigned to it. To do that, we execute the `SQL_IS_GROUP_EMPTY` query, with the `groupName` dynamically inserted. If the number of records returned is greater than zero, the group can't be deleted, and an exception is thrown, which, remember, will be displayed to the user courtesy of the DWR exception-handler function we set up. Assuming the group is devoid of users, the deletion can go ahead, and we once more return the updated list of groups to the UI.

ReportDescriptor.java

The `ReportDescriptor` class is a simple VO that stores information about a given report job that has been created in the portal. Its UML diagram can be seen here in Figure 7-21.

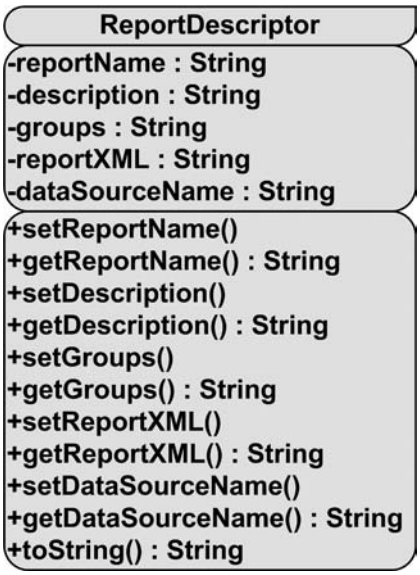


Figure 7-21. *UML diagram of the* `ReportDescriptor` *class*

No code is shown here as it's just a collection of private fields with public getters and setters for each.

ReportWorker.java

The `ReportWorker` class is very much like the `FavoritesWorker` and `GroupWorker` class we've looked at already, but as obvious as the fact that Worf is apparently like that ugly kid in school who somehow still managed to get the best girls just because he seemed dangerous (come on, Dax **and** Deanna?!?), the `ReportWorker` deals with reports. You can see its UML diagram in Figure 7-22.

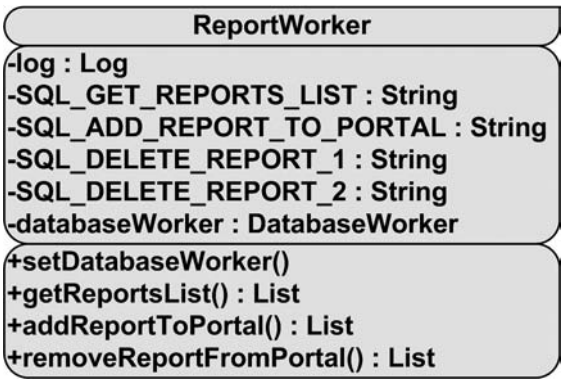


Figure 7-22. *UML diagram of the* `ReportWorker` *class*

There are four SQL queries present in this class, and they are listed and described in Table 7-6.

Table 7-6. *The SQL Statements Contained in the* `ReportWorker` *Class*

SQL Statement	Class Field	Description
`SELECT * FROM reports`	`SQL_GET_REPORTS_LIST`	Returns a list of reports known to the portal
`INSERT INTO reports (reportname, description, groups, reportxml, datasourcename) VALUES('${reportname}', '${description}', '${groups}', '${reportxml}', '${datasourcename}')`	`SQL_ADD_REPORT_TO_PORTAL`	Adds a report to the portal
`DELETE FROM reports WHERE reportname='${reportname}'`	`SQL_DELETE_REPORT_1`	Deletes a report from the portal (first of two queries, this one for the `reports` table)
`DELETE FROM schedules WHERE reportname='${reportname}'`	`SQL_DELETE_REPORT_2`	Deletes a report from the portal (second of two queries, this one for the `schedules` table)

The first method we find is `getReportsList()`, which actually is a bit different from the other list methods we've seen before.

```
@SuppressWarnings("unchecked")
public List<ReportDescriptor> getReportsList(
  final HttpServletRequest inRequest) throws Exception {

  try {

    List<Map> reports = databaseWorker.executeQuery(SQL_GET_REPORTS_LIST,
      new HashMap());
    List<ReportDescriptor> reportList = new ArrayList<ReportDescriptor>();

    for (Map m : reports) {
      boolean userCanSeeReport = false;
      String reportGroups = (String)m.get("GROUPS");
      if (reportGroups.indexOf("AllUsers") != -1) {
        userCanSeeReport = true;
      } else {
        List<String> groupsUserIsIn =
          (ArrayList)inRequest.getSession().getAttribute("groups");
        if (groupsUserIsIn != null) {
          for (String s : groupsUserIsIn) {
            if (reportGroups.indexOf(s) != -1) {
              userCanSeeReport = true;
```

```
            }
          }
        }
      }

      if (userCanSeeReport) {
        ReportDescriptor report = new ReportDescriptor();
        report.setReportName((String)m.get("REPORTNAME"));
        report.setDescription((String)m.get("DESCRIPTION"));
        report.setGroups(reportGroups);
        report.setReportXML((String)m.get("REPORTXML"));
        reportList.add(report);
      }
    }
    return reportList;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

First, we have a query to get the list of reports, nothing too different there. Then, however, we see some interesting bits in the loop structure. First, for each report, we look in the groups field and see whether the value `AllUsers` is present. If it is, the `userCanSeeReport` flag is set to `true`, so the report can be shown in the various report lists on the screen. Second, we get the list of groups the user belongs to from session. This is an `ArrayList` containing the name of each group the user belongs to (he or she can belong to more than one, remember). For each, we do an `indexOf()` check to see whether that group name appears in the list of groups for the report. If it does, we set `userCanSeeReport` to `true`. Finally, when that is complete, we check the value of `userCanSeeReport`. If it's `true`, we instantiate and populate a `ReportDescriptor` object, and add it to a `List` that is returned. Therefore, the outcome of all this is that only reports a user is allowed to see by virtue of being a member of the group or groups that the reports have been granted to will be available to the user in any of the report lists in the UI.

The next method is, as I'm sure you suspect, the `addReportToPortal()` method:

```
public List<ReportDescriptor> addReportToPortal(
  final ReportDescriptor inReport, final HttpServletRequest inRequest)
  throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", inReport.getReportName());
    tokens.put("description", inReport.getDescription());
    tokens.put("groups", inReport.getGroups());
```

```
    tokens.put("reportxml", inReport.getReportXML());
    tokens.put("datasourcename", inReport.getDataSourceName());

    databaseWorker.executeUpdate(SQL_ADD_REPORT_TO_PORTAL, tokens);
    return getReportsList(inRequest);

  } catch (DataIntegrityViolationException dive) {
    throw new Exception("Report could not be created.\n\n" +
      "(Does the report already exist?)");
  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Since what is passed in to this method is a `ReportDescriptor` object, we simply pull the values we need from it for inclusion in the data `Map` to replace tokens in the SQL query with, and off we go. Note the same `DataIntegrityViolationException` handling as we saw earlier. Also note that this method, and a number of others, has the `HttpServletRequest` as a parameter. Recall that DWR will automagically populate this for us, and it's time we have such a parameter present.

Removing a report from the portal is the next function, and it is encapsulated in the `removeReportFromPortal()` method:

```
public List<ReportDescriptor> removeReportFromPortal(
  final String inReportName, final HttpServletRequest inRequest)
  throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", inReportName);

    databaseWorker.executeUpdate(SQL_DELETE_REPORT_1, tokens);
    databaseWorker.executeUpdate(SQL_DELETE_REPORT_2, tokens);
    return getReportsList(inRequest);

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

Like all its predecessor removal methods, it works in the same basic fashion, except for one detail: to delete a report, it not only needs to be removed from the `reports` table, but also the `schedules` table. That's the reason there are two queries being executed here.

UserDescriptor.java

The `UserDescriptor` class is a simple VO that stores information about a given user that has been created in the portal. Its UML diagram can be seen here in Figure 7-23.

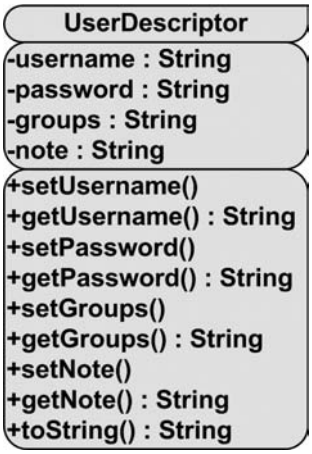


Figure 7-23. *UML diagram of the* `UserDescriptor` *class*

No code is shown here as it's just a collection of private fields with public getters and setters for each.

UserWorker.java

As I did for the user functions when we reviewed `RePortal.js`, I'm going to skip the vast majority of `UserWorker` on the grounds that if you simply review `GroupWorker`, and then change the word "Group" to "User" everywhere, you've basically reviewed `UserWorker` too! However, there is one method we do need to take a look at, and it just happens to be the corollary to the `logUserIn()` method we saw in `RePortal.js`.

Before that though, take a look at the UML diagram for the `UserWorker` class, shown in Figure 7-24.

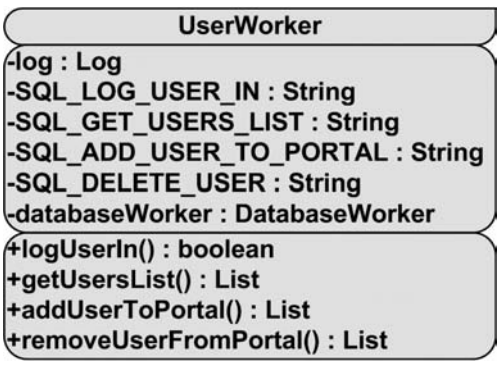


Figure 7-24. *UML diagram of the* `UserWorker` *class*

And just to be at least somewhat thorough, the SQL queries present in this class are shown in Table 7-7.

Table 7-7. *The SQL Statements Contained in the* `UserWorker` *Class*

SQL Statement	Class Field	Description
`SELECT * FROM users WHERE username='${username}' AND password='${password}'`	`SQL_LOG_USER_IN`	Attempts to log a user in by trying to find a match for the given username and password
`SELECT * FROM users`	`SQL_GET_USERS_LIST`	Returns the list of users on the portal
`INSERT INTO users (username, password, groups, note) VALUES('${username}', '${password}', '${groups}', '${note}')`	`SQL_ADD_USER_TO_PORTAL`	Adds a user to the portal
`DELETE FROM users WHERE username='${username}'`	`SQL_DELETE_USER`	Removes a user from the portal

Now, as for that one method, it's `logUserIn()` here in `UserWorker` too, so let's see what it does now:

```
public boolean logUserIn(final String inUsername, final String inPassword,
  final HttpServletRequest inRequest) throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("username", inUsername);
    tokens.put("password", inPassword);

    List users = databaseWorker.executeQuery(SQL_LOG_USER_IN, tokens);

    if (users.size() == 1) {
      inRequest.getSession(true).setAttribute("username", inUsername);
      Map m = (Map)users.get(0);
      String groupsString = (String)m.get("GROUPS");
      StringTokenizer st = new StringTokenizer(groupsString, ",");
      List<String> groups = new ArrayList<String>();
      while (st.hasMoreTokens()) {
        String nextGroup = st.nextToken();
        groups.add(nextGroup);
      }
      inRequest.getSession(true).setAttribute("groups", groups);
      return true;
    } else {
      return false;
    }
```

```
  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

So, first a query is done that attempts to retrieve the user's record from the `users` table based on username and password. If a single match is found, the user is logged in. The username is added as a session attribute, which lets the app be reloaded, and the user won't have to log in again because of that code we saw on the client with the username being passed in to `init()`. Next, we get the list of groups the user belongs to, which is a comma-separated string. The string is tokenized, and each token (a group name) is added to a `List`, which is then added to session under the key `groups`. Lastly, `true` is returned, which signals to the UI code that the user successfully logged in. If the record isn't found for this user in the `users` table, `false` is returned.

And that wraps up the `UserWorker` class!

ReportRunner.java

Now we come to a couple of classes that are a fair bit different from the other `xxxWorker` classes we've looked at so far, the first of which is the `ReportRunner` class. This class is responsible for actually running reports. This is used not only for ad hoc runs requested by the user, but also by the scheduled jobs that Quartz runs. Because of that, this class extends from the Quartz `Job` class, which mandates a single method . . . but I'm getting ahead of myself a bit!

In Figure 7-25, you can see the UML diagram for this class.

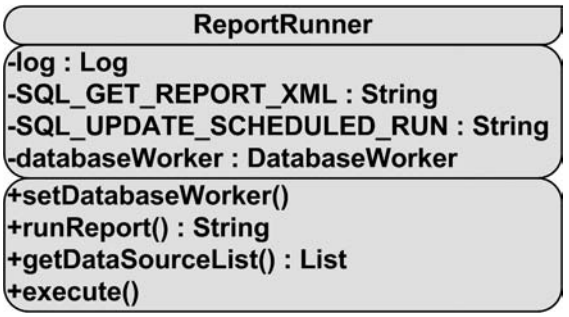


Figure 7-25. *UML diagram of the* `ReportRunner` *class*

And Table 7-8 lists the SQL queries used in this class.

Table 7-8. *The SQL Statements Contained in the* `ReportRunner` *Class*

SQL Statement	Class Field	Description
`SELECT datasourcename, reportxml FROM reports where reportname='${reportname}'`	`SQL_GET_REPORT_XML`	Gets the XML that is the report definition
`UPDATE schedules SET lastrun='${lastrun}', lastrunstatus='${lastrunstatus}', lastrunoutput='${lastrunoutput}' WHERE reportname='${reportname}' AND daysofweek='${daysofweek}' AND runtime='${runtime}'`	`SQL_UPDATE_SCHEDULED_RUN`	Updates the database when a report has been run

Now, before we get into the code, let's be a little graphically oriented and take a look at the output from a report execution. I freely admit this isn't the fanciest report you're ever going to see, but it **is** an actual, live, working report. In Figure 7-26, you can see this output.

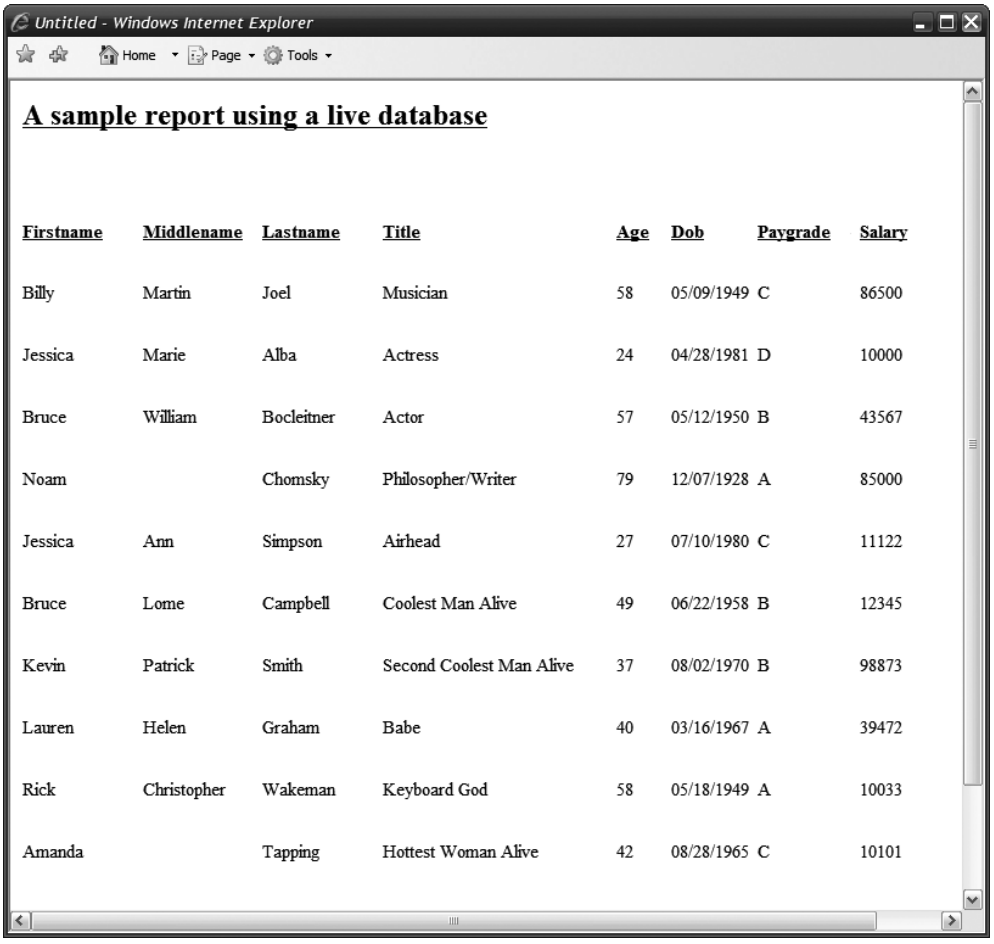

Untitled - Windows Internet Explorer
Home
Page
Tools
A sample report using a live database
Firstname Middlename Lastname Title Age Dob Paygrade Salary
Billy Martin Joel Musician 58 05/09/1949 C 86500
Jessica Marie Alba Actress 24 04/28/1981 D 10000
Bruce William Bocleitner Actor 57 05/12/1950 B 43567
Noam Chomsky Philosopher/Writer 79 12/07/1928 A 85000
Jessica Ann Simpson Airhead 27 07/10/1980 C 11122
Bruce Lome Campbell Coolest Man Alive 49 06/22/1958 B 12345
Kevin Patrick Smith Second Coolest Man Alive 37 08/02/1970 B 98873
Lauren Helen Graham Babe 40 03/16/1967 A 39472
Rick Christopher Wakeman Keyboard God 58 05/18/1949 A 10033
Amanda Tapping Hottest Woman Alive 42 08/28/1965 C 10101

Figure 7-26. *The results of running a report*

The output of all reports in RePortal is HTML, even though DataVision supports other formats (and in fact, some of the other layout engines, such as the PDF layout engine, produce output a little better than either of the two HTML layout engines . . . however, with HTML, there are no external dependencies and no browser plug-ins required, so HTML is a nice, safe choice in a book situation like this where you just want the example to work out of the box).

The method that generates this output (indirectly) is the aptly named `runReport()` method, as seen here:

```
public String runReport(final String inReportName) throws Exception {

  Connection conn = null;

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", inReportName);

    List resultSet = databaseWorker.executeQuery(SQL_GET_REPORT_XML, tokens);

    Map m = (Map)resultSet.get(0);
    String dataSourceName = (String)m.get("DATASOURCENAME");
    String reportXML = (String)m.get("REPORTXML");

    DataSourceDescriptor dsd = Config.getDataSource(dataSourceName);
    String jdbcDriverClassName = dsd.getDatabaseDriver();
    String connectionInfo = dsd.getDatabaseURI();
    String userName = dsd.getDatabaseUsername();
    String password = dsd.getDatabasePassword();
    Class.forName(jdbcDriverClassName);
    conn = DriverManager.getConnection(connectionInfo, userName, password);

    Report report = new Report();
    report.setDatabaseConnection(conn);
    StringReader sbis = new StringReader(reportXML);
    report.read(new org.xml.sax.InputSource(sbis));
    StringWriter sw = new StringWriter();
    report.setLayoutEngine(new CSSHTMLLE(new PrintWriter(sw)));

    report.runReport();
    return sw.toString();

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  } finally {
    if (conn != null) {
      try {
        conn.close();
```

```
      } catch (Exception e) {
        // Eat this exception, for the sake of simplicity.
      }
    }
  }

}
```

The first task this method performs is to get the XML for the report named by the `inReportName` argument. The query that gets this also returns the name of the data source the report is to run against. With that, a call to `Config.getDataSource()` can be made, which returns a `DataSourceDescriptor` object. That object contains all the information we need to get a connection to the database, and that's precisely what is done next.

After that comes the actual report generation code. Using DataVision is a piece of cake. First, a `Report` object is instantiated. We then hand the database connection we just created to it. Next, we get a `StringReader` on the string containing the report's XML. After that, we ask the `Report` instance to read the XML using an `InputSource`. After that, a new `StringWriter` is instantiated, which is where our output will wind up. Then, we set the desired layout engine, which is what DataVision uses to generate our output. In this case, we're using the `CSSHTMLLE` layout engine, which generates HTML using CSS (there is a plain `HTMLLE` layout engine as well that uses a table-based approach . . . the CSS-based engine produces results that are generally better). Finally, a call to the `runReport()` method of the `Report` object actually runs the report and writes the output to the `StringWriter`. All that's left is to return a `String` version of that `StringWriter`. Also note the `finally` block that cleans up the database connection, which is nothing out of the ordinary for JDBC programming.

Next up is a quick-and-easy method named `getDataSourceList()`. This is responsible for supplying the UI with a list of data sources to display.

```
public List<DataSourceDescriptor> getDataSourceList() throws Exception {

  try {

    List<DataSourceDescriptor> dataSources =
      new ArrayList<DataSourceDescriptor>();
    Map<String, DataSourceDescriptor> cds = Config.getDataSources();
    for (String dsKey : cds.keySet()) {
      dataSources.add(cds.get(dsKey));
    }
    return dataSources;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

I'm quite sure you can figure that one out on your own, so exercise that positronic net of yours!

The last method in this class is the `execute()` method, and it's the method responsible for allowing this class to be used by Quartz, since it fulfills the `Job` interface contract.

```
public void execute(JobExecutionContext inContext)
  throws JobExecutionException {

  log.trace("execute() - Entry");

  String reportName = "unknown";

  try {

    // Get name of report to run from job detail.
    JobDataMap dataMap = inContext.getJobDetail().getJobDataMap();
    reportName = dataMap.getString("reportName");
    if (log.isDebugEnabled()) {
      log.debug("execute() - reportName = " + reportName);
    }

    // We need a DatabaseWorker here to do the work.
    databaseWorker = new DatabaseWorker();

    // Run the report and get the generated HTML.
    String reportOutput = runReport(reportName);

    // Write the report output to the database. Start by creating a map of
    // replacement tokens for SQL statement.
    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", reportName);
    tokens.put("daysofweek", dataMap.getString("daysOfWeek"));
    tokens.put("runtime", dataMap.getString("runTime"));
    tokens.put("lastrunoutput", reportOutput);
    tokens.put("lastrunstatus", "OK");
    tokens.put("lastrun", new Date().toString());
    databaseWorker.executeUpdate(SQL_UPDATE_SCHEDULED_RUN, tokens);

  } catch (Exception e) {
    log.error("execute() - Exception(1): " + e);
    try {
      Map<String, String> tokens = new HashMap<String, String>();
      tokens.put("reportname", reportName);
```

```
        tokens.put("runtime", "-");
        tokens.put("daysofweek", "-");
        tokens.put("lastrunoutput", "");
        tokens.put("lastrunstatus", "ERROR");
        tokens.put("lastrun", e.getMessage());
        databaseWorker.executeUpdate(SQL_UPDATE_SCHEDULED_RUN, tokens);
        throw new JobExecutionException(e.getMessage());
      } catch (Exception e1) {
        log.error("execute() - Exception(2): " + e1);
        throw new JobExecutionException(e1.getMessage());
      }
    }

    log.trace("execute() - Exit");

} // End execute().
```

When Quartz calls this in response to a report job trigger firing, it passes in a `JobExecutionContext` object, which is essentially a container for information the job may need to execute. From this object we grab the `JobDataMap`, which is simply a `Map` containing information we populated when the job was added (you'll see this code in the `ReportSchedulingWorker` class, which is yet to come). Within this `Map` is the name of the report, which is what we need to do the required work, so it is retrieved.

Next, a `DatabaseWorker` instance is created. Note that this instance is set at class level. Let's think this through . . . typically, this class will be called by DWR, so Spring will be injecting a `DatabaseWorker` instance for us. However, when Quartz instantiates this class and calls `execute()`, Spring isn't in the mix, so how do we get a `DatabaseWorker` instance, since it will be needed? Fortunately, there's nothing to stop us from instantiating it ourselves and setting the class-level reference to it, which results in basically the same thing as what Spring does for us, so we're good to go after that.

A call to `runReport()` is then made to actually execute the report, and the resultant HTML output is stored. Next, the database record for this scheduled job is updated, including storing that report output so it is viewable from the UI.

After that is a `catch` block to deal with any problems. An attempt is made to write out the exception message to the database, but a secondary `catch` is below that to deal with a failure during that update.

ReportScheduleDescriptor.java

The `ReportScheduleDescriptor` class is a simple VO that stores information about a given scheduled report job that has been created in the portal. Its UML diagram can be seen here in Figure 7-27.

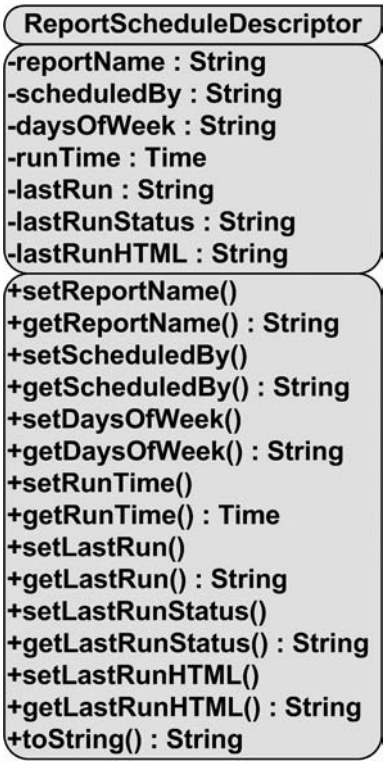


Figure 7-27. *UML diagram of the* `ReportScheduleDescriptor` *class*

No code is shown here as it's just a collection of private fields with public getters and setters for each.

ReportSchedulingWorker.java

Now, we can finally see the light at the end of the tunnel as there is only one class left to explore! Here now, in Figure 7-28, you can see the UML of that final class, `ReportSchedulingWorker`, which is responsible for listing, adding, and deleting scheduled report jobs; viewing the generated output of a given report execution; and interfacing with Quartz.

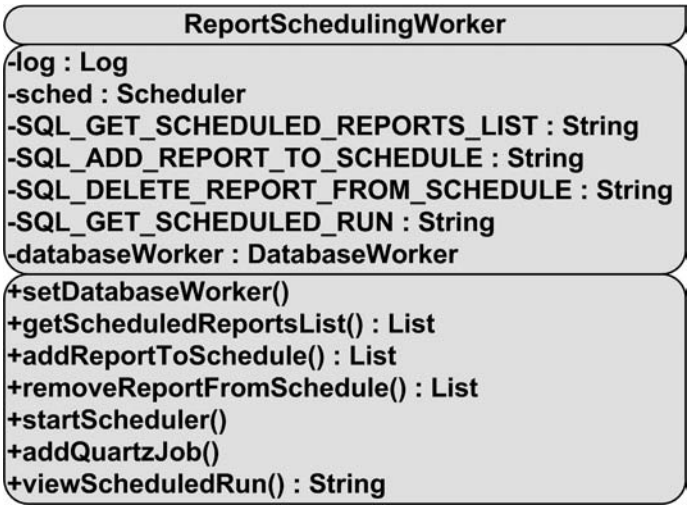


Figure 7-28. *UML diagram of the* `ReportSchedulingWorker` *class*

A handful of SQL statements are required by this class to function, and those statements are summarized here in Table 7-9.

Table 7-9. *The SQL Statements Contained in the* `ReportSchedulingWorker` *Class*

SQL Statement	Class Field	Description
`SELECT * FROM schedules`	`SQL_GET_SCHEDULED_REPORTS_LIST`	Returns the list of reports scheduled in the portal
`INSERT INTO schedules (reportname, scheduledby, daysofweek, runtime) VALUES('${reportname}', '${scheduledby}', '${daysofweek}', '${runtime}')`	`SQL_ADD_REPORT_TO_SCHEDULE`	Adds a report to the list of scheduled reports
`DELETE FROM schedules WHERE reportname='${reportname}' AND daysofweek='${daysofweek}' AND runtime='${runtime}'`	`SQL_DELETE_REPORT_FROM_SCHEDULE`	Removes a report from the list of scheduled reports
`SELECT lastrunoutput FROM schedules WHERE reportname='${reportname}' AND daysofweek='${daysofweek}' AND runtime='${runtime}'`	`SQL_GET_SCHEDULED_RUN`	Returns the HTML generated by the specified report job run

The first method to examine is `getScheduledReportsList()`, shown here:

```
@SuppressWarnings("unchecked")
public List<ReportScheduleDescriptor> getScheduledReportsList()
  throws Exception  {

  try {

    List<Map> reports = databaseWorker.executeQuery(
      SQL_GET_SCHEDULED_REPORTS_LIST, new HashMap());
    List<ReportScheduleDescriptor> reportList =
      new ArrayList<ReportScheduleDescriptor>();

    for (Map m : reports) {
      ReportScheduleDescriptor report = new ReportScheduleDescriptor();
      report.setReportName((String)m.get("REPORTNAME"));
      report.setScheduledBy((String)m.get("SCHEDULEDBY"));
      report.setDaysOfWeek((String)m.get("DAYSOFWEEK"));
      report.setRunTime((Time)m.get("RUNTIME"));
      report.setLastRun((String)m.get("LASTRUN"));
      report.setLastRunStatus((String)m.get("LASTRUNSTATUS"));
      reportList.add(report);
    }
    return reportList;

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

It's very much like all the other list methods we've seen, so there's no need to go into it, I think. Let's instead get right to the next method, `addReportToSchedule()`:

```
@SuppressWarnings("unchecked")
public List<ReportScheduleDescriptor> addReportToSchedule(
  final ReportScheduleDescriptor inReportScheduleDescriptor)
  throws Exception {

  try {

    Map tokens = new HashMap();
    tokens.put("reportname", inReportScheduleDescriptor.getReportName());
    tokens.put("scheduledby", inReportScheduleDescriptor.getScheduledBy());
    tokens.put("daysofweek", inReportScheduleDescriptor.getDaysOfWeek());
    tokens.put("runtime", inReportScheduleDescriptor.getRunTime().toString());
```

```
    databaseWorker.executeUpdate(SQL_ADD_REPORT_TO_SCHEDULE, tokens);
    addQuartzJob(inReportScheduleDescriptor);
    return getScheduledReportsList();

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

It starts out like all the other methods used to add items to the database with the typical data `Map` creation and SQL execution. It even ends like all the others with a call to the list method, `getScheduledReportsList()` this time, and the result of that method call is returned for the UI to update the display.

In between those two things though is something unique, namely the call to `addQuartzJob()`. As the name implies, that method is responsible for adding a new scheduled job to the Quartz scheduler. The code for the method is this:

```
private void addQuartzJob(final ReportScheduleDescriptor inRSD)
  throws Exception {

  GregorianCalendar gc = new GregorianCalendar();
  gc.setTime(inRSD.getRunTime());

  StringTokenizer st = new StringTokenizer(inRSD.getDaysOfWeek(), ",");
  while (st.hasMoreTokens()) {
    String nextDay = st.nextToken();
    String reportName = inRSD.getReportName() + gc.get(Calendar.HOUR_OF_DAY) +
      gc.get(Calendar.MINUTE) + nextDay;

    JobDetail jobDetail =
      new JobDetail(reportName, null, ReportRunner.class);
    JobDataMap dataMap = jobDetail.getJobDataMap();
    dataMap.put("reportName", inRSD.getReportName());
    dataMap.put("runTime", inRSD.getRunTime().toString());
    dataMap.put("daysOfWeek", inRSD.getDaysOfWeek());

    int day = 0;
    if (nextDay.equalsIgnoreCase("SUN")) { day = 1; }
    if (nextDay.equalsIgnoreCase("MON")) { day = 2; }
    if (nextDay.equalsIgnoreCase("TUE")) { day = 3; }
    if (nextDay.equalsIgnoreCase("WED")) { day = 4; }
    if (nextDay.equalsIgnoreCase("THU")) { day = 5; }
    if (nextDay.equalsIgnoreCase("FRI")) { day = 6; }
    if (nextDay.equalsIgnoreCase("SAT")) { day = 7; }
```

```
    Trigger trigger = TriggerUtils.makeWeeklyTrigger(day,
      gc.get(Calendar.HOUR_OF_DAY), gc.get(Calendar.MINUTE));
    trigger.setStartTime(new Date());
    trigger.setName(reportName + "_Trigger");

    sched.scheduleJob(jobDetail, trigger);
  }

}
```

First, a `Calendar` (more specifically, a subclass of `Calendar`, `GregorianCalendar`) is instantiated and its time set to the time specified by the incoming `ReportScheduleDescriptor` object. Next, the comma-separated list of days of the week the report should run is tokenized. We then iterate over all the tokens. Here's where it gets slightly bizarre . . . because I've chosen to use a weekly trigger for the scheduled job, that means that for every day the report is scheduled to run, we actually need to schedule a separate job. Because of the way the UI is designed, you can only schedule any given report to run once per day, and since we can pick and choose what days to run on, the weekly trigger makes sense. But, a scheduled job has to be assigned a unique name. That unique name is the combination of the name of the report, the hour and minute the report is scheduled to run, and the day of the week. That's what is being constructed with the line

```
String reportName = inRSD.getReportName() + gc.get(Calendar.HOUR_OF_DAY) +
  gc.get(Calendar.MINUTE) + nextDay;
```

Following that a `JobDetail` object is created, and within it goes a `JobDataMap` object. Into this map goes all the information about the report being run.

Next, a quick conversion is performed to get a numeric value for the day of the week being scheduled.

Finally, a `Trigger` is instantiated and constructed, which is really what schedules the report to run on the specified day at the specified time. We set the trigger to start immediately by calling `trigger.setStartTime(new Date());`, we give the `Trigger` itself a unique name based on the constructed unique report name, and finally we call `scheduleJob()` on the `Scheduler` instance, and we're done.

Removing a report from the scheduler is also a pretty easy exercise and is rather similar to adding it:

```
public List<ReportScheduleDescriptor> removeReportFromSchedule(
  final String inReportName, final String inDaysOfWeek,
  final String inRunTime) throws Exception {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", inReportName);
    tokens.put("daysofweek", inDaysOfWeek);
    tokens.put("runtime", inRunTime);

    databaseWorker.executeUpdate(SQL_DELETE_REPORT_FROM_SCHEDULE, tokens);
```

```
    StringTokenizer st = new StringTokenizer(inRunTime, ":");
    String hour = st.nextToken();
    String minute = st.nextToken();
    st = new StringTokenizer(inDaysOfWeek, ",");
    while (st.hasMoreTokens()) {
      String day = st.nextToken();
      String reportName = inReportName + hour + minute + day;
      sched.deleteJob(reportName, null));
    }

    return getScheduledReportsList();

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

First, we delete the record from the database. Next, we construct the same unique report name as we did when adding the report, but notice the logic is a little different because this time we have the time the report is scheduled to run in the form `xx:yy`, so it's a little bit easier to do because it's just some basic string manipulations. Finally, a call to `deleteJob()` on the `Scheduler` object removes the job from Quartz. That's that!

The next method to check out is `startScheduler()`, which is called from the `ContextListener` when the application starts up.

```
public void startScheduler() throws Exception {

  SchedulerFactory schedFact = new StdSchedulerFactory();
  sched = schedFact.getScheduler();
  sched.start();

  List<ReportScheduleDescriptor> scheduledReports = getScheduledReportsList();
  for (ReportScheduleDescriptor rsd : scheduledReports) {
    addQuartzJob(rsd);
  }

}
```

This is really pretty simple. First, a `Scheduler` instance is retrieved from the `SchedulerFactory` provided by Quartz. Next, the `Scheduler` is started. Finally, we retrieve the list of scheduled reports from the database by calling the `getScheduledReportsList()` method, and then for each we'll call the `addQuartzJob()` method, passing it the `ReportScheduleDescriptor` object.

Only one method remains between us and completion of our five-year mission . . . err, maybe not quite **that** long, but you get the idea! Anyway, that method is `viewScheduledRun()`, and it is this bit of code:

```
@SuppressWarnings("unchecked")
public String viewScheduledRun(
  final String inReportName, final String inDaysOfWeek,
  final String inRunTime) throws Exception {

  try {

    Map<String, String> tokens = new HashMap<String, String>();
    tokens.put("reportname", inReportName);
    tokens.put("daysofweek", inDaysOfWeek);
    tokens.put("runtime", inRunTime);

    List<Map> records = databaseWorker.executeQuery(
      SQL_GET_SCHEDULED_RUN, tokens);
    return (String)((Map)records.get(0)).get("LASTRUNOUTPUT");

  } catch (Exception e) {
    e.printStackTrace();
    throw e;
  }

}
```

This method is called when the user selects a report that has run from the list of scheduled reports, and it is responsible for returning the HTML generated when the report last executed. To do so, we take the input arguments, which you'll notice is the same information that is used in the previous two methods to create that unique report name the report was scheduled under in Quartz, which again is essentially a composite key for the appropriate record in the database. With that information thrown into a `Map`, we can execute the SQL query to get the `LASTRUNOUTPUT` field and return it. The UI does its thing and displays it, and that's the full cycle for the scheduled report view functionality.

Whew, that's finally it, we're done! I hope you enjoyed this chapter—both the project and the numerous *Star Trek* references.

Suggested Exercises

There are a number of suggestions I can think of for this project, as there are definitely plenty that can be done to make it better. Here are just a few, all of which would prove to be excellent learning experiences for you:

- I purposely left one extremely easy thing undone specifically so you'd have a quickie to get your feet wet: password change! It's a little inconvenient to have to delete a user and re-create that user with a new password, wouldn't you say? So, give the user the ability to change his or her own password. This should be little more than one new method on the `UserWorker` and the associated UI changes (I'd suggest playing with the lightbox stuff for this). Once you do this, you can move on to some of the more involved suggestions that follow.

- You'll notice one of the limitations of RePortal is that only reports with no parameters can be run. While this is probably good enough for a simplistic dashboard-type portal, it's clearly not good enough to be a more robust report portal like you'd want in a true enterprise environment. So, one suggestion is to allow for parameters to be entered and used by the report. This will require parsing the report XML (using Commons Digester perhaps?) and looking for the parameters (explore the DataVision report format; it's not very complicated at all). You'll most likely want to add a `getReportParameters()` method to the `ReportRunner` class, which will be called when the user clicks a report from his or her favorites. This should return a `List` of `ReportParamDescriptor` objects, and you can then use this list to generate an HTML form the user can fill out. Collect the parameters and pass them along to the `runReport()` method. The details of using parameters with DataVision reports programmatically can be found in my article on using DataVision in a web application here: `www.omnytex.com/articles`. You'll find that it amounts to not much more than one or two simple method calls on the `Report` object. This may sound like a lot of work, but I think you'll find it's really not too much, and it'll definitely get you some more experience with DWR, not to mention DataVision.
- Another limitation of RePortal that is pretty apparent is the need to delete groups, users, and reports to make changes to them. Therefore, I suggest adding the ability to simply edit existing items. The only reason this limitation is present is that the code size was getting a little too big for a book like this, and this was one limitation I could live with, and it frankly left me with a decent suggestion for you that'll let you exercise your DWR chops a bit more. It shouldn't be a whole lot more than adding the appropriate `updateXXX()` method to the `GroupWorker`, `UserWorker`, and `ReportWorker` classes, plus the necessary client-side code to execute it (and to put the data in an editable form to begin with).
- The groups in this application really don't serve as much purpose as they probably should. It wouldn't be anything unusual if only a single group could maintain users. Perhaps only a single group should be allowed to add reports. You get the idea! I therefore suggest adding the ability to specify rights that a group has. You could also go a step further and do the same for users, and allow the users' rights to override that of the group.

Don't worry if some of these suggestions seem a little tough . . . I leave you with the immortal words of Korris, as spoken to Worf in the *Next Generation* episode "Heart of Glory": "Do not deny the challenge of your destiny! Get off your knees and soar. Open your eyes and let the dream take flight."

Summary

In this (very long) chapter, we touched on quite a few things. First, you learned some new DWR functionality in the form of Spring integration, and a bunch of new functions available in `dwr.util`. You learned a bit more about Spring, namely its IoC (Dependency Injection) capabilities. We also touched on a number of third-party libraries including DataVision, Commons Digester, Quartz, and script.aculo.us.

And ultimately, we built what I believe is a rather useful application. Not perfect, of course (that's what the suggested exercises are for!), but pretty useful out of the box.

We also, of course, had our daily dose of classic science fiction, and perhaps a few other pop-culture references mixed in too.

In the next chapter, we'll take a little break from all this enterprise-type development . . . we'll get into some Bermuda shorts, a comfy T-shirt, and sneakers, and code ourselves up a little game.

Before that though, I close this chapter the way I opened it: with a *Star Trek* cliché:

"Beam me up, Scotty!"

(Ok, even **I** hate myself for making that obvious joke!)

CHAPTER 8

DWR for Fun and Profit (a DWR Game!)

So, here we are, just two chapters to go before we reach the end. Before we continue on though, let's take stock of where we've been. So far, we've built a webmail client, a wiki, a file manager, and a reporting portal. All of these are, more or less, "enterprise-y" kinds of applications; certainly they are meant to be serious apps, tools to get a job done.

But you know what they say about all work and no play (makes Homer something . . . something[1]), so for the sake of not turning into axe-wielding maniacs bent on killing our loved ones, let's break the pattern this time around and have ourselves some fun.

And what better project could there be to do that than a game? If you've read either of my two previous books, you know that each has a game project, and this book will be no different, but it's not just because games are fun to make and play (although both those things are certainly true).

As it turns out, a game is a great project to show off some cool DWR features, namely reverse Ajax. Games in general are excellent programming exercises because they tend to touch on so many disciplines and present unique challenges to overcome. They usually are harder than you think too. For instance, if you want to do a web-based game as we're going to do here, what parts of it run client side vs. server side? How will network latency factor into it? All of these things, and more, go into a web-based game project.

But, in typical fashion, DWR has ways of making it all just far easier than it probably has any right to be. So, what kind of game is it? Let's see . . .

Application Requirements and Goals

Choosing a game to develop was actually a bit challenging because I knew I wanted to demonstrate reverse Ajax with this project, but what kind of game is good for this? Well, something turn-based sprang to mind immediately, perhaps Battleship? Yeah, that would work great! Problem is, the DWR folks already saw fit to use that idea. Argh. OK, plan B then . . . how about some sort of card game? Playing poker against a server sounds pretty good. The problem there

1. In *The Simpsons* "Treehouse of Terror V" episode, in the parody of *The Shining*, Homer goes crazy, just like the original Jack Nicholson character, and tries to murder Marge, Bart, Lisa, and Maggie, but winds up freezing outside with the rest of the family.

though is that the logic for a halfway decent opponent is a little tough to code. Well, for someone not all that good at poker to begin with anyway, it is!

So, then I thought: what about dots? Sure, that could work. But then it occurred to me that to really show off reverse Ajax requires something a little more real time, not quite turn based.

So, to make a long story short, I decided that playing the classic game of Memory against the server would be a decent idea. Simply put, you and the server would try to clear your own board at the same time, and whoever finished first wins. It's a pretty simple concept, but it winds up being a good project to do with DWR.

So, let's talk specifics:

- We'll show two grids of tiles on the screen, each grid being 7×6 tiles in size, and you as the human try to clear one grid while the server tries to clear the other.
- The server will need to make its moves and report back to the client what to put on the screen, hence the need for reverse Ajax.
- Just for some added fanciness, when two matching tiles are found, we won't simply remove them from the screen, we'll destroy them in a puff of fire, otherwise known as an explosion!
- I've told you that DWR can be configured with annotations as well, so how about we see that in action this time around?
- In an admittedly contrived circumstance, we'll have a "need" to write our own custom DWR `Creator` in this project.

That's not a big list of requirements, but then, it's not inherently a complex game! We aren't creating Halo 3 here after all! That's by design actually, because after the large size of the last chapter, I thought something a little less . . . well, just **less**! . . . would be a welcome change.

So, off to the races we go!

DWR Annotations

Throughout this book, I've used the configuration-file-driven method of configuring DWR. You can achieve the same thing, assuming you're using Java 5 or higher, with annotations. For instance, we'll see later that there is a remotable class called `GameCore` in this project. At the start of the class you'll see this:

```
@RemoteProxy
public class GameCore {
```

You have to tell DWR that the class itself is remotable, and the `@RemoteProxy` annotation does exactly that. By default, the name of the JavaScript object matches the class, but should you want to name it differently, as you would do with the `javascript` attribute of the `<create>` tag in `dwr.xml` (which you'll recall from earlier discussions allows you to give any name you like to the JavaScript proxy stub version of your remotable server-side class), you would simply use

```
@RemoteProxy(name="GameCore")
```

Although it isn't used in this project, there is another class-level annotation available, that being

```
@DataTransferObject
```

This is used when you have a class that will be returned from a remote method, and it does the same job as the `<convert>` tag in `dwr.xml` does, telling DWR that this class can be converted to JavaScript and returned. This uses the bean converter by default, but if you need to specify a different one, you can use

```
@DataTransferObject(converter="something_else")
```

Another piece to this puzzle is that any field you wish to return as part of the object must be annotated with another annotation:

```
@RemoteProperty
```

Going back to the `@RemoteProxy` annotation case though . . . once you've annotated your class as being remotable, you also have to annotate each method that should be remotable, and you do this like so:

```
@RemoteMethod
public String myMethod() { return ""; }
```

Any method not annotated in this will not be available in the remote proxy object (a.k.a. the JavaScript stub of this class that you call from your code).

There is one other step in turning annotation support on, and that involves some differences in `web.xml`. We'll see that when we examine that file; it's really not too big a deal.

Reverse Ajax in Action

In Chapter 3, I described the reverse Ajax support DWR offers, but at that point I didn't go into the details of how it's used. Now is the time for that!

Reverse Ajax in DWR works basically by some server process writing out JavaScript function calls. These essentially get stored in a queue, and when the time comes, they are sent to the browser. Now, "when the time comes" has different meanings depending on what type of reverse Ajax you've decided to use. For instance, if you use the inactive piggyback method, these calls will be sent as part of the next incoming request. If you're using Comet, they will be sent (more or less) in real time.

So, how do we actually go about doing this? It's actually quite simple:

```
WebContext webContext = WebContextFactory.get();
ScriptBuffer script = new ScriptBuffer();
script.appendScript("someJavascriptFunction();");
ScriptSession scriptSession = webContext.getScriptSession();
scriptSession.addScript(script);
```

Yep, that's about all it takes! The JavaScript function `someJavascriptFunction()` will be executed at the next transmission from the server to the client, which again depends on what type of reverse Ajax you're using. In this project, we're using Comet, so it's pretty much real time.

The `WebContext` object, in essence, allows the server-side code to connect to the current page for a given user (or group of users). Through it we can write out JavaScript function calls that will be sent to the browser(s) for execution.

While I think DWR is a fantastic project and is generally well documented, I've unfortunately found the documentation of annotations and especially reverse Ajax to not quite be at the same level as other facets of DWR. I spent a fair amount of time digging through the source code of DWR itself and trying to find some simplistic samples scattered around the Web to figure this all out. There are a couple of examples that come with DWR itself of reverse Ajax, but it's not commented especially well and it's not as informative as I'd like it to be. My hope is that this project will help fill the gaps just a little bit though. Besides, I couldn't get through this *entire* book without *some* sort of criticism, could I?

It may take a few moments to get your brain wrapped around all this. The first time I ran the application and saw this work, I was frankly a little surprised! It goes against much of what you've always thought you knew about the way communication between a client and a server works on the Web. Of course, a few more moments of thought, and you quickly realize everything actually **is** working just like you always knew it did, it's just that there is a really nice illusion in front of it. In any case, it allows you to do some things that you otherwise couldn't do, and that's cool in my book!

Anything Else, or Can We Get Goin' Already?!?

Since there aren't any support libraries to discuss this time around, there's not really anything in the way of preparation before beginning to actually dissect this project beyond the topic of annotations and reverse Ajax. The only other new topic here is writing our own custom creator. That, however, will make more sense if seen in context, so it probably wouldn't help much to show a preview of that here. Instead, let's just jump right in, feet first, and take a quick peek at the application, as shown in Figure 8-1.

When the application first loads, the tiles on both play grids randomly cycle through all the different tile images that you will be trying to match. By the way, these tiles are all in-game graphics from my PocketPC game K&G Arcade (`www.omnytex.com/products.shtml`). This game contains 25 mini-games, one of them a memory game called InMemoria. So now you know where the name comes from. (FYI, if you have a PocketPC, K&G Arcade, as well as Invasion: Trivia!, the other game featuring nobody's favorite wisecracking aliens Krelmac and Gentoo, are now free from Clickgamer when you sign up for the ClickPayGo service. Check it out at `www.clickgamer.com`.)

When you are ready to play, I'll give you just one guess which button you click!

From there out, it's a simple matter of clicking two tiles and seeing whether they match. If not, after a brief delay, they flip back over. If they match, they explode off the grid. All the while, the grid on the right is being played by an invisible opponent, namely the server the application is running on.

Like I said, this isn't exactly Grand Theft Auto, but it can be kind of fun (my son and daughter enjoyed play-testing it, but it's challenging enough for us adults too, I promise!).

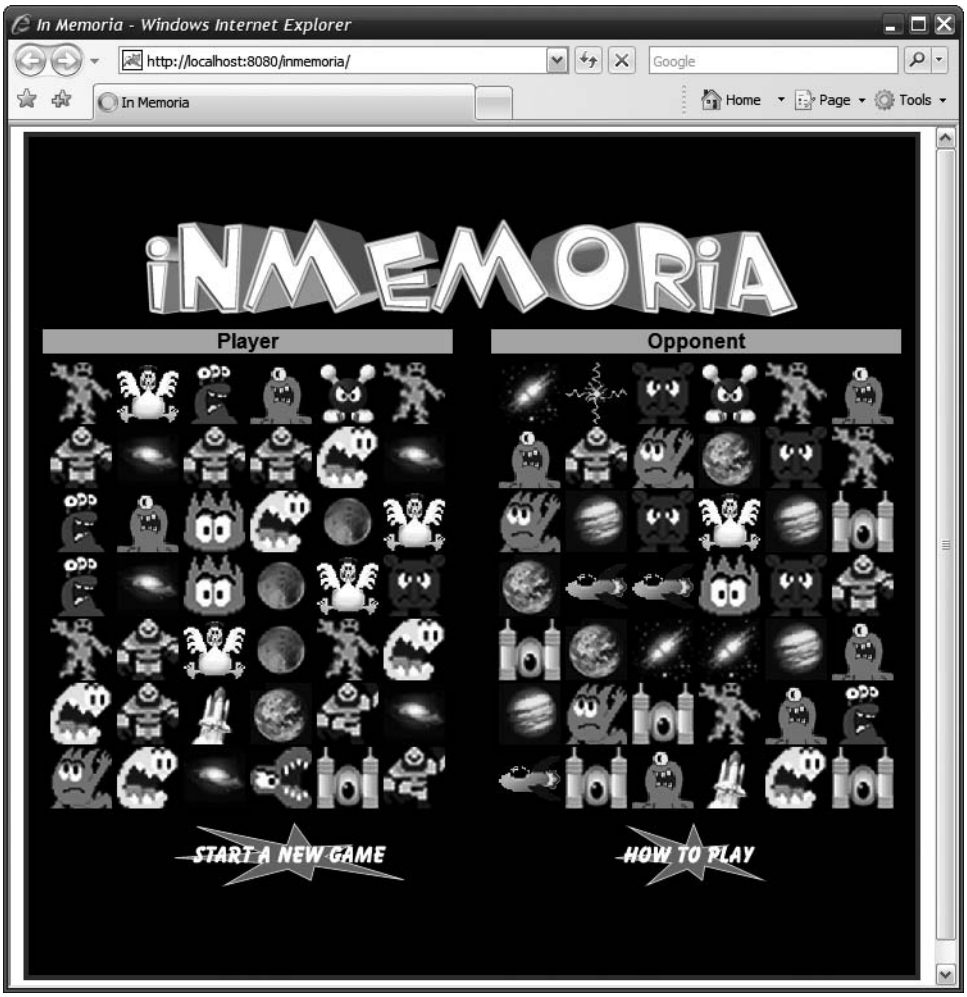


Figure 8-1. *The InMemoria screen as it appears shortly after startup (the tiles have randomly cycled a little here)—this definitely looks better in color, so play the game already!*

Dissecting InMemoria

(With the Muppets singing backing vocals): Movin' right along . . . we begin by having a look at the directory structure of the application, which is shown in Figure 8-2.

Of course, we have the normal, typical, boring, everyday Java webapp directory structure here. We find the usual `styles.css` file in the `css` directory. We also find an `img` directory where all the images are stored (not expanded here). The `js` directory has a single file, `InMemoria.js`, where all our client-side logic resides. In the root directory we have two files, `howToPlay.txt`, which contains the help file seen when you click the How To Play button, and the single, main file for the client-side of the game, `index.jsp`.

That's it, simplicity itself!

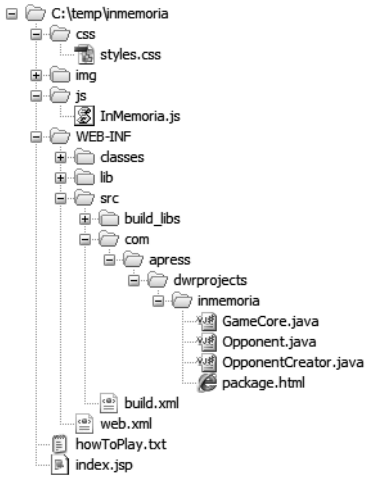


Figure 8-2. *The directory structure of the application*

Configuration Files

One thing that you should immediately notice is that there is no `dwr.xml` file in sight! That makes sense when you remember one of our requirements: use annotations to configure this application. So, there's only a single configuration file to worry about, and that's `web.xml`.

web.xml

The `web.xml` file here is substantially similar to all the previous ones, but there is some difference in the servlet configuration because you need to use a different DWR servlet when you're going to be using annotation-based configuration. In Listing 8-1, you can see that the servlet class is no longer `uk.ltd.getahead.dwr.DWRServlet`, as we've been using all along, but is now `org.directwebremoting.servlet.DwrServlet`.

Listing 8-1. *The* `web.xml` *File*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app id="game" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee➥
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Game</display-name>

  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
    <init-param>
```

```
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>activeReverseAjaxEnabled</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
       <param-name>classes</param-name>
       <param-value>
          com.apress.dwrprojects.inmemoria.GameCore,➥
com.apress.dwrprojects.inmemoria.Opponent
       </param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>

  <!-- Session timeout config. -->
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>

  <!-- Welcome file config. -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

You'll also quickly notice two new parameters, namely `activeReverseAjaxEnabled`, which, as I'm sure you can guess, turns reverse Ajax support on. More specifically, it uses the so-called "active" variants of reverse Ajax, namely polling or Comet (a.k.a. long-lived HTTP requests). If you were to set this to `false`, you would enable the "inactive" method, called *piggybacking*, where the server waits for an incoming request and then adds the information that is to be pushed out to the client in an opportunistic sort of way.

The second parameter that's new here is the one actually related to annotations, namely the `classes` parameter. For every class you intend to configure with annotations, you have to list it here. I actually found this to be a little counterintuitive in that if I'm using annotations, I don't then expect to have to put anything in a configuration file. While `web.xml` perhaps holds a bit of a special place as a configuration file since it's required for a basic Java webapp, it still

feels like a little bit more work than should be required. Be that as it may though, these are the requirements, and so we'll just cope and carry on! It's not like it's an overly huge burden anyway, right?

The Client-Side Code

Well, exploring the configuration files was fun, but it didn't take that long! Now it's time to move on to the client-side code, which also frankly isn't all that voluminous. Our first stop along that great journey is the style sheet file.

styles.css

The `styles.css` file, shown in all its glory in Listing 8-2, holds no surprises frankly, but it's here nonetheless.

Listing 8-2. *The Extremely Run-of-the-Mill Style Sheet,* `styles.css`

```
/* Catch-all style. */
* {
  font-family      : arial;
  font-size        : 12pt;
  font-weight      : bold;
  color            : #ffff00;
}


/* Style of document body. */
.cssBody {
  background-color : #ffffff;
  margin           : 4px;
  padding          : 0px;
}


/* Style of container that makes up screen. */
.cssContainer {
  background-color : #000000;
  border           : 4px solid #ff0000;
}


/* Style of playfield grid headers. */
.cssGridHeader {
  background-color : #ffa0a0;
  color            : #000000;
}
```

```
/* Style of divider between the two grids. */
.cssGridDivider {
  height           : 6px;
}
```

We have the wildcard selector to give us a consistent font across the entire application. Then there is `cssBody`, which applies to the `<body>` of the document. Its only purpose really is to ensure we have some space around the main container (the black portion) that the game lives in. Then, the `cssContainer` selector applies to that container itself, and it's just a black background with a solid red border. The `cssGridHeader` selector covers the text above both of the playfields indicating which side is the human player and which is the computerized opponent. Finally, the `cssGridDivider` just gives us a style to use on the table cell between the two grids to specify how big that gap is.

If any of that was surprising or new to you, I highly suggest getting a good book on Cascading Style Sheets because it's all pretty basic, simple stuff.

index.jsp

Just like the style sheet file, the main file that initially loads, `index.jsp`, as shown in Listing 8-3, is very small, simplistic, and shouldn't hold any surprises for you.

Listing 8-3. *The Complete Markup for the Game Page, Encapsulated in* `index.jsp`

```
<html>
  <head>
    <title>In Memoria</title>

    <link rel="StyleSheet" href="css/styles.css" type="text/css">
    <script type="text/javascript" src="js/InMemoria.js"></script>

    <!-- DWR imports. -->
    <script src="dwr/engine.js"></script>
    <script src="dwr/util.js"></script>
    <script src="dwr/interface/GameCore.js"></script>
    <script src="dwr/interface/Opponent.js"></script>

    <!-- Button image preloads. -->
    <script>
      var btnStart0 = new Image(179, 50);
      var btnStart1 = new Image(179, 50);
      btnStart0.src = "img/btnStart0.gif";
      btnStart1.src = "img/btnStart1.gif";
      var btnHowToPlay0 = new Image(120, 50);
      var btnHowToPlay1 = new Image(120, 50);
      btnHowToPlay0.src = "img/btnHowToPlay0.gif";
      btnHowToPlay1.src = "img/btnHowToPlay1.gif";
    </script>
```

```
  </head>

  <body onLoad="inMemoria.init();" class="cssBody">

    <table cellpadding="0" cellspacing="0" align="center" height="100%"
      width="700" height="500"><tr><td align="center" valign="middle"
        class="cssContainer">

      <!-- Title. -->
      <div><img src="img/title.gif" vspace="10"></div>

      <!-- Play grids. -->
      <table border="0" cellpadding="0" cellspacing="0">
        <tr>
          <td align="center" class="cssGridHeader">Player</td>
          <td> </td>
          <td align="center" class="cssGridHeader">Opponent</td>
        </tr>
        <tr><td class="cssGridDivider"></td></tr>
        <tr>
          <td width="320" align="center" valign="middle" id="divPlayer"></td>
          <td width="30"> </td>
          <td width="320" align="center" valign="middle" id="divOpponent"></td>
        </tr>
      </table>

      <!-- Buttons. -->
      <img src="img/btnStart0.gif" hspace="80" vspace="10"
        onClick="inMemoria.startGame();"
        onMouseOver="this.style.cursor='pointer';this.src=btnStart1.src;"
        onMouseOut="this.style.cursor='';this.src=btnStart0.src;">
      <img src="img/btnHowToPlay0.gif" hspace="80" vspace="10"
        onClick="inMemoria.howToPlay();"
        onMouseOver="this.style.cursor='pointer';this.src=btnHowToPlay1.src;"
        onMouseOut="this.style.cursor='';this.src=btnHowToPlay0.src;">

    </td></tr></table>

  </body>

</html>
```

We have the usual style sheet, core game logic (`InMemoria.js`), and DWR imports in the `<head>`, followed by some image preloads. These are used for the two buttons so we aren't calling the server on each mouse event (at least until browser caching kicks in).

On document load, we call `inMemoria.init()`, which kick-starts everything, as we'll see very shortly. Beyond that, it's really just a whole bunch of plain old HTML.

One interesting point though is that the two playfield grids are not built here; that is, you might expect to see a whole bunch of `<img>` tags here, but you don't. As it turns out, they are built dynamically for no other reason than 84 `<img>` tags would look pretty ugly (the grid is 7×6, and there's two grids, so 84 total tiles).

In Figure 8-3, you can see what the screen looks like right after the Start A New Game button has been clicked, right before the first tiles are selected by either player or computer (I snapped this screenshot quickly, as the computer makes its first move immediately).

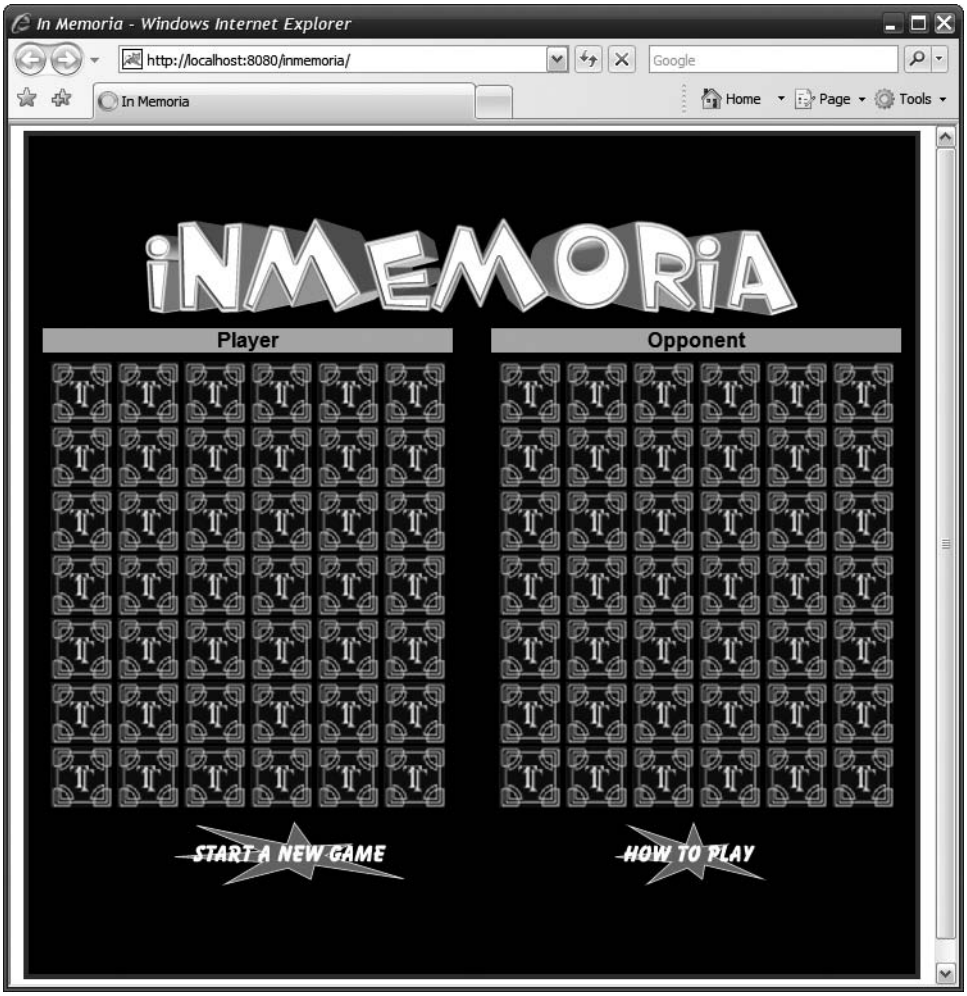


Figure 8-3. *Here's a look at the game as it starts before any tiles have been selected by the player or the computer opponent.*

Of course, aside from the Start A New Game button, there's the How To Play button (as if anyone actually needs it!), and that leads to the `howToPlay.txt` file being displayed, which is convenient since that's our next stop . . .

howToPlay.txt

I haven't shown the listing of this file here because it is nothing but a plain text file, so it's just as easy, and probably prettier, to show what it looks like on the screen, and Figure 8-4 is exactly that.

Windows Internet Explorer
Welcome to InMemoria!
Playing the game is simple. Begin by clicking the "Start A New Game" button. Then, simply click two tiles on the Player playfield grid trying to find two that match. When you find them, they will explode and be cleared. If the two you chose are not a match, they will flip back over. Play continues until you clear all tiles.
Of course, it's not *quite* that simple! The server will be playing against you, and the thing is pretty good! If it succeeds in clearing all its tiles before you, then you lose.
So come on, beat that machine, the human race is counting on you!
OK

Figure 8-4. *The immensely impressive help dialog box (bwuahahah!)*

As I said earlier, it's unlikely too many people actually need help to understand how to play this game, but there was an ulterior motive to my putting this function in here, and that was to show DWR's ability to read from other URLs and return the results. We'll see what that's all about when we get to the server-side code, but suffice it to say it's the mechanism that allows us to make a DWR Ajax call, read this text file (which could have just as easily been a JSP), and display it.

InMemoria.js

Moving right along, we now find ourselves reaching our first bit of real, actual, *interesting* code, and that's the client-side core logic, found in the `InMemoria.js` file.

Gimme Some Data, Baby!

Within this file we find the `InMemoria` class, and that class begins with a healthy serving of data fields, which are summarized in Table 8-1.

Table 8-1. *The Data Fields Found in the* `InMemoria` *Class*

Field Name	Description
`tileImages`	This is an array of `Image` objects, one for each tile type.
`explosionImages`	This is an array of `Image` objects, one for each animation frame that makes up the explosion animation cycle.
`noGameInProgressInterval`	This is a pointer to a JavaScript interval (timer) that continually fires while no game is in progress to randomly show different tile images.
`playerProcessClicks`	This is a flag that indicates whether player tile clicks are being processed (this is `false` when no game is in progress, and also when nonmatching tiles are being unflipped or when matching tiles are being exploded).

Field Name	Description
playerTileValues	This is an array where each element is one of the tiles in the player's grid describing what tile image is found there.
playerFlippedTile1	This is the tile number of the first tile the player currently has flipped.
playerFlippedTile2	This is the tile number of the second tile the player currently has flipped.
playerExplosionInterval	This is a pointer to a JavaScript interval that is used to do the explosion animation cycle for the player, one tick per frame.
playerExplosionFrame	This is the current animation frame being shown for exploding tiles for the player.
playerExplosionDir	This indicates whether we are incrementing or decrementing an index into the explosionImages array, in other words, whether the explosion is expanding or contracting for the player.
opponentFlippedTile1	This is the tile number of the first tile the computer opponent currently has flipped.
opponentFlippedTile2	This is the tile number of the second tile the computer opponent currently has flipped.
opponentExplosionInterval	This is a pointer to a JavaScript interval that is used to do the explosion animation cycle for the opponent, one tick per frame.
opponentExplosionFrame	This is the current animation frame being shown for exploding tiles for the opponent.
opponentExplosionDir	This indicates whether we are incrementing or decrementing an index into the explosionImages array, in other words, whether the explosion is expanding or contracting for the opponent.

First Things First: init()

The first method we come across in this class is init(), which we saw in index.jsp is called upon page load. Here's that code now:

```
this.init = function() {

  // Kick off DWR reverse-ajax (Comet).
  dwr.engine.setActiveReverseAjax(true);

  // Preload tile images. 0 is always unflipped.
  var img = new Image(48, 48);
  img.src = "img/tileUnflipped.gif";
  this.tileImages.push(img);
  for (var i = 1; i < 22; i++) {
    img = new Image(48, 48);
    img.src = "img/tile" + i + ".gif";
    this.tileImages.push(img);
  }
```

```
  // Preload explosion images.
  for (var i = 0; i < 5; i++) {
    img = new Image(48, 48);
    img.src = "img/explosion" + i + ".gif";
    this.explosionImages.push(img);
  }

  // Create player and opponent grids.
  var refPlayer = dwr.util.byId("divPlayer");
  var refOpponent = dwr.util.byId("divOpponent");
  var tileIndex = 1;
  for (var y = 0; y < 7; y++) {
    for (var x = 0; x < 6; x++) {
      // Player tile.
      img = document.createElement("img");
      img.setAttribute("id", "tilePlayer_" + tileIndex);
      img.setAttribute("src", "img/tileUnflipped.gif");
      img.setAttribute("hspace", "2");
      img.setAttribute("vspace", "1");
      img.setAttribute("style", "cursor:pointer");
      refPlayer.appendChild(img);
      // Opponent tile.
      img.onclick = inMemoria.tileClick;
      img = document.createElement("img");
      img.setAttribute("id", "tileOpponent_" + tileIndex);
      img.setAttribute("src", "img/tileUnflipped.gif");
      img.setAttribute("hspace", "2");
      img.setAttribute("vspace", "1");
      refOpponent.appendChild(img);
      tileIndex++;
    }
    refPlayer.appendChild(document.createElement("br"));
    refOpponent.appendChild(document.createElement("br"));
  }

  // Start interval for randomly cycling tiles waiting for game to start.
  inMemoria.randomlyCycleTiles();

} // End init().
```

The first thing we need to do is the other half of the equation to turn DWR's reverse Ajax capabilities on, and that's telling the client-side DWR engine to turn it on with the line `dwr.engine.setActiveReverseAjax(true)`. If you're running the game in Firefox and you have Firebug open (or a similar tool such as HTTPWatch in Internet Explorer), at this point you'll see a request start up that doesn't seem to end. That's the Comet connection to the server. Making use of this connection is a snap, but it's a server-side function, so we'll get to that in a bit.

After that, we have two chunks of code that do image preloads, one of all the tile images and the other the frames of animation for the tile explosion cycle. Both of these populate an array of `Image` objects.

Next we see the code that creates those tiles that you'll recall weren't in the markup in `index.jsp`. It's really nothing more than creating some `<img>` tag objects, setting their attributes appropriately, and appending them to the DOM. Note that the player grid and the opponent grid are both populated at the same time.

The last thing done in this method is a call to the `randomlyCycleTiles()` method. That's the next thing we're going to look at, and it is this bit of code:

```
this.randomlyCycleTiles = function() {

if (this.noGameInProgressInterval == null) {

  this.noGameInProgressInterval =
       setInterval("inMemoria.randomlyCycleTiles()", 500);

} else {

  // Randomly choose a tile image for every tile and show it.
  for (var i = 1; i < 43; i++) {
       // Do player grid first, and then opponent grid.
       var tile = null;
       var newValue = null;
       // Only update the next tile 50% of the time.
       if (Math.round(0 + (1 - 0) * Math.random()) == 1) {
         // Choose one of the 21 tile types.
         tile = dwr.util.byId("tilePlayer_" + i);
         newValue = Math.round(1 + (21 - 1) * Math.random());
         tile.src = this.tileImages[newValue].src;
       }
       // Only update the next tile 50% of the time.
       if (Math.round(0 + (1 - 0) * Math.random()) == 1) {
         // Choose one of the 21 tile types.
         tile = dwr.util.byId("tileOpponent_" + i);
         newValue = Math.round(1 + (21 - 1) * Math.random());
         tile.src = this.tileImages[newValue].src;
       }
  }

}

} // End randomlyCycleTiles().
```

This is, of course, the code responsible for showing the different tile images randomly as the application waits for the user to start a game. It works like this: upon the first call to it, the `noGameInProgressInterval` field will be null, and so a call to `setInterval()` will set up a

repeating call to this method, set to fire every half-second (500 milliseconds). Upon subsequent calls, the `else` part of the `if` block executes, and the actual work is done.

The "actual work" in this case amounts to cycling through all the tiles in both grids. For each, we first randomly decide whether we're going to update that tile or not. That's where the line `if (Math.round(0 + (1 - 0) * Math.random()) == 1) {` comes into play. The `if` clause winds up picking either a `0` or `1` randomly, and only if it's a `1` will the tile be updated. This equates to a 50 percent probability of updating the tile with each iteration of the interval.

Assuming the logic encapsulated in the `if` statement decides to update a given tile, it's then just a matter of a random choice of any of the 21 different tile images and an update of the appropriate `<img>` element's `src` attribute, pointing it to the `src` of the new tile image, which is stored in that `tileImages` array we saw earlier. This process is then repeated for the opponent's grid, and that's that.

And the Kick Is Up: startGame()

Next is the method `startGame()`, which is called when an error occurs.

No, I kid, I kid. It's really, and obviously I think, the method called when the Start A New Game button is clicked. I don't want to keep you in suspense, so here it is:

```
this.startGame = function() {

// Call server to get randomized grids, and kick off the game.
GameCore.startGame( {
  callback : function(inResp) {

      // Stop the little effect seen while waiting for a game to start.
      clearInterval(inMemoria.noGameInProgressInterval);

      // Set tiles for player grid and reset states and images to unflipped.
      inMemoria.playerTileValues = new Array();
      inMemoria.playerTileValues.push(0);
      for (var i = 1; i < 43; i++) {
        inMemoria.playerTileValues.push(inResp[i - 1]);
        inMemoria.playerFlippedTile1 = null;
        inMemoria.playerFlippedTile2 = null;
        var tile = dwr.util.byId("tilePlayer_" + i);
        tile.src = inMemoria.tileImages[0].src;
        tile.style.visibility = "visible";
      }

      // Now do the same for the opponent grid.
      for (var i = 1; i < 43; i++) {
        var tile = dwr.util.byId("tileOpponent_" + i);
        tile.src = inMemoria.tileImages[0].src;
        tile.style.visibility = "visible";
      }
```

```
      // Begin processing player tile clicks.
      inMemoria.playerProcessClicks = true;

  }
} );

} // End startGame().
```

Starting a game isn't too tough . . . first, the `startGame()` method of the server-side `GameCore` class is called. We'll look at this later, but for now it's enough to know that it returns an array of tile values telling the client code what images apply to each tile for the player's grid. Note that the opponent's grid is stored client-side only. Upon return of this call, the array returned by the server is transferred into the `playerTileValues` field, the `playerFlippedTile1` and `playerFlippedTile2` fields are cleared, and all of the player's tiles are changed to the unflipped image. The same is done for the opponent's grid (as far as changing to the unflipped image anyway . . . the `opponentFlippedTile1` and `opponentFlippedTile2` fields are not reset because it isn't necessary, as they get set later anyway when the opponent picks tiles). The final task is to set the `playerProcessClicks` field to `true` so that the player can begin clicking tiles, and at that point a game has officially begun!

Wouldn't Be a Tile-Matching Game Without It: tileClick()

The next method you'll find in the `InMemoria` class is the largest and most complex, and it's the code that handles when the player clicks a tile, the `tileClick()` method:

```
this.tileClick = function() {

// Abort if player clicks are blocked (during explosions and unflipping).
if (!inMemoria.playerProcessClicks) { return; }

// Get index of clicked tile.
var tileIndex = this.id.split("_")[1];

// No tile is currently flipped...
if (inMemoria.playerFlippedTile1 == null) {

  // Set image based on tile value.
  this.src =
       inMemoria.tileImages[inMemoria.playerTileValues[tileIndex]].src;
  // Record currently flipped.
  inMemoria.playerFlippedTile1 = tileIndex;

// OK, a tile is already flipped, so...
} else {

  // Is the tile already flipped the one that was clicked?
  if (inMemoria.playerFlippedTile1 == tileIndex) {
```

```
        // Yes it was...unflip this, and any other flipped tile.
        inMemoria.unFlip(inMemoria.playerFlippedTile1,
          inMemoria.playerFlippedTile2, "Player");

    } else {

        // Nope, two tiles are now effectively flipped. First, show this one
        // as flipped too and block any further player clicks for now.
        inMemoria.playerProcessClicks = false;
        inMemoria.playerFlippedTile2 = tileIndex;
        this.src =
          inMemoria.tileImages[inMemoria.playerTileValues[tileIndex]].src;

        // Now, check for a match.
        if (inMemoria.playerTileValues[inMemoria.playerFlippedTile1] ==
          inMemoria.playerTileValues[inMemoria.playerFlippedTile2]) {

          // Match, so do explosion.
          inMemoria.playerExplosionFrame = 0;
          inMemoria.playerExplosionDir = true;
          inMemoria.playerExplosionInterval =
               setInterval("inMemoria.cyclePlayerExplosion()", 100);

        } else {

          // No match, so pause for half a second and unflip them both.
          setTimeout("inMemoria.unFlip(inMemoria.playerFlippedTile1, " +
               "inMemoria.playerFlippedTile2, 'Player');", 500);

        }

    }

}

} // End tileClick().
```

So, it's the longest method in this class, but it really isn't all that difficult to follow, I think. As we do so, keep in mind that this method is "attached" as the `onClick` handler to all the tiles in the player's grid. This will be important to keep in mind because it gives you the correct context of the `this` keyword.

First, we do a quick rejection: if the `playerProcessClicks` field isn't set to `true`, we abort this method by returning immediately from it. This takes care of the player trying to click tiles before a game is actually in progress, and also when nonmatching tiles are being unflipped or when matching tiles are being exploded. Next, we take the value of the `id` attribute of this tile and split it on the underscore character. If you look back at the `init()` method, you'll see that the `id` attribute is in the form `tilePlayer_xx`, where `xx` is the number of the tile. It also happens to be the index into the `playerTileValues` array, so by doing this split and getting the

second element of the array, the part following the underscore, we can examine the correct array element that corresponds to this tile.

Next is another decision: is there already a tile currently flipped? If not, then all we really need to do is show the correct image for the tile, which is a simple matter of getting the value from the `playerTileValues` array for this tile, and using that as the index into the `tileImages` array, and setting the `src` attribute of this tile to the `src` attribute of the `Image` object in the array. We then store this tile's index in the `playerFlippedTile1` field, and we're done for now.

The other possibility is that there is already a tile flipped. In that case, we have a second decision to make: is this tile that was just clicked the one that was previously flipped? If so, we unflip it by calling the handy-dandy `unFlip()` method, which is this bit of code:

```
this.unFlip = function(inTile1, inTile2, inPlayerOrOpponent) {

if (inTile1 != null) {
  dwr.util.byId("tile" + inPlayerOrOpponent + "_" + inTile1).src =
       inMemoria.tileImages[0].src;
}
if (inTile2 != null) {
  dwr.util.byId("tile" + inPlayerOrOpponent + "_" + inTile2).src =
       inMemoria.tileImages[0].src;
}
if (inPlayerOrOpponent == "Player") {
  inMemoria.playerFlippedTile1 = null;
  inMemoria.playerFlippedTile2 = null;
  inMemoria.playerProcessClicks = true;
}

} // End unFlip().
```

The arguments of this method indicate which tiles are flipped and whether we're unflipping player tiles or opponent tiles. Unflipping the tiles is simply a matter of changing their `src` attribute to the unflipped tile image. Also, if we're unflipping player tiles, we clear the two fields that record what tiles are flipped and ensure that player clicks will be processed again. By making this a separate method with this structure, we can use this to unflip both player and opponent tiles, and it allows us to do this generically whether two matching tiles were just found, two tiles that don't match are being unflipped, or a tile is already unflipped because the player clicked it twice in a row.

With `unflip()` out of the way, we can pick up where we left off in `tileClick()`. Where we left off was the second `else` branch, which corresponds to the situation where a second tile is clicked that isn't the already-flipped tile. In that case, two tiles are logically flipped at this point. So, we need to determine whether the player found a match. First, we block further player clicks by setting `playerProcessClicks` to `false`. Next, we flip over the tile that was just clicked. Third, we compare their values from the `playerTileValues` array. If they match, we begin the explosion of the tiles by starting off with the first animation frame and ensuring we are incrementing the frame number each iteration. Finally, we set an interval to fire every tenth of a second and call the `cyclePlayerExplosion()` method.

If the tiles don't match, we let the player see them both for half a second by setting a timeout. Remember, the difference between a timeout and an interval is that a timeout fires once

after a defined period of time, while an interval fires continuously on a specified schedule. Once the timeout elapses, it calls `unFlip()`, which you recall sets `playerProcessClicks` to `true`, so at that point the player can make another selection.

Makin' It Go Boom: cyclePlayerExplosion()

Let's jump back to the `cyclePlayerExplosion()` method I mentioned a paragraph or so ago. The code in that method is here:

```
this.cyclePlayerExplosion = function() {

  if (inMemoria.playerExplosionDir) {

    // Increment to next frame.
    inMemoria.playerExplosionFrame++;
    // Reverse direction when it's time.
    if (inMemoria.playerExplosionFrame == 4) {
      inMemoria.playerExplosionDir = false;
    }

  } else {

    // Decrement to next frame.
    inMemoria.playerExplosionFrame--;
    // When animation cycle is done, remove tiles and reset vars.
    if (inMemoria.playerExplosionFrame < 0) {
      clearInterval(inMemoria.playerExplosionInterval);
      dwr.util.byId("tilePlayer_" +
        inMemoria.playerFlippedTile1).style.visibility = "hidden";
      dwr.util.byId("tilePlayer_" +
        inMemoria.playerFlippedTile2).style.visibility = "hidden";
      inMemoria.playerFlippedTile1 = null;
      inMemoria.playerFlippedTile2 = null;
      inMemoria.playerProcessClicks = true;
      // Call server to check for win.
      inMemoria.checkForWin();
      return;
    }

  }

  // Show explosion frame on both flipped tiles.
  dwr.util.byId("tilePlayer_" +
    inMemoria.playerFlippedTile1).src =
    inMemoria.explosionImages[inMemoria.playerExplosionFrame].src;
  dwr.util.byId("tilePlayer_" +
```

```
      inMemoria.playerFlippedTile2).src =
      inMemoria.explosionImages[inMemoria.playerExplosionFrame].src;

  } // End cyclePlayerExplosion().
```

The way an explosion here works is simple. There are five frames of animation. A complete explosion animation cycle is displaying all of them in ascending order, and then displaying all of them in descending order. This gives the illusion of expanding hot gas (i.e., an explosion!), which then contracts and dissipates. Each frame is 1/10 of a second in length, or 100 milliseconds. So, another interval is set up with that period that fires this method each iteration.

So first, this method looks at the `playerExplosionDir` field. When it's `true`, we're incrementing the frame number each time. So, all we need to do each time is check to see what frame we're on. If we're on frame 4 (it's zero-based, so frame 4 is the fifth frame), we set `playerExplosionDir` to `false`, which reverses things, meaning we'll begin to decrement the frame number.

Now, when `playerExplosionDir` is false, the logic is similar. Decrement the frame number, and then check it to see whether we're at less than zero. When we are, the explosion is done. At that point, the two tiles are hidden, and variables are reset to ensure the player can now make another selection free and clear. The method is exited, and play can continue.

Note that except for the case where the explosion cycle has finished, the code falls through to two lines at the end that display the appropriate animation frame on both tiles being exploded.

If you're curious what an explosion looks like, Figure 8-5 shows you simultaneous explosions occurring on both the player and opponent's grids.

Is It Time to Gloat Yet?—checkForWin()

I left out one thing in the preceding explanation, and that's the call to `checkForWin()` that occurs when the explosion cycle completes. This of course checks to see whether the player has won the game, and it's the following code:

```
  this.checkForWin = function() {

    GameCore.checkForWin( {
      callback : function(inResp) {
        if (inResp) {
          alert("Congratulations, you won!");
        }
      }
    } );

  } // End checkForWin().
```

Yep, I kid you not, that's it! The server is what actually determines whether the player has won or not, and we'll see how that works when we're done with all the client-side code, which is actually very soon now!

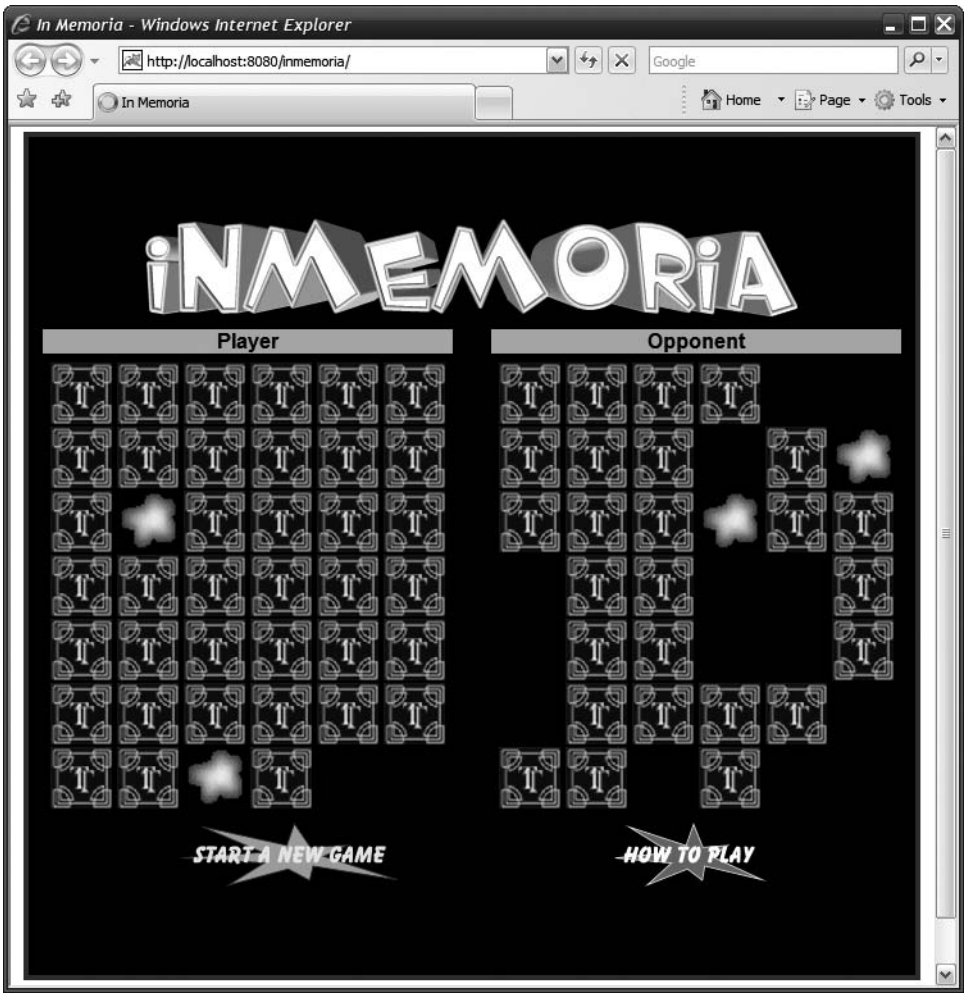


Figure 8-5. *Oh look, the player and opponent have found a match at the same time!*

I skipped a method there, namely the `cycleOpponentExplosion()` method. It's analogous to the `cyclePlayerExplosion()` method, so I won't go over it here. Take a look and you'll see though that it's very much similar. One difference you'll see is that instead of `checkForWin()` being called at the end, a method called `outcomeComplete()` is called on a different class, namely the `Opponent` class. All shall be revealed there, have no fear!

The next four methods are actually a little bit difficult to explain without looking at the server-side code because they are in fact called **from** the server! They are executed in conjunction with the reverse Ajax capability. Rather than explore them right now, I'm going to skip them and come back to them as we look at the server side. So, with that, only a single method remains, and that's `howToPlay()`:

```
this.howToPlay = function() {

GameCore.howToPlay( {
  callback : function(inResp) {
       alert(inResp);
  }
} );

} // End howToPlay().
```

As I'm sure you can surmise, the `howToPlay()` method of the `GameCore` class on the server returns a string, and also as I'm sure you can guess, it's the contents of the `howToPlay.txt` file. Now, exactly *how* that file is returned is of interest, but it's of course on the server side.

Fortunately, you won't have to wait too long for that explanation because we're now ready to move on to the server-side code!

The Server-Side Code

The server side of InMemoria consists of a whopping *three* classes. Two of them are remotable, at least *parts* of them are, and the third is an extension to DWR itself, using one of DWR's known extension points. Let's jump right in with the `GameCore` class that you've seen called a number of times from the client code.

GameCore.java

In Figure 8-6, you can see the rather tiny class UML diagram for the `GameCore` class. In this case, three methods, the public ones, are remotable, while the fourth private method is not.

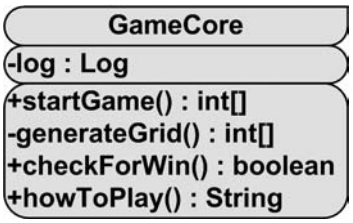


Figure 8-6. *UML class diagram of the* `GameCore` *class*

The first method is `startGame()`, which we saw earlier is called from the similarly named `startGame()` method in the client-side `InMemoria` class. Its code is this:

```
@RemoteMethod
public int[] startGame(final HttpServletRequest inRequest) {

  log.trace("startGame() - Entry");

  try {
```

```
    // Create random set of tile data for player and opponent alike.
    int[] userGrid = generateGrid();
    int[] opponentGrid = generateGrid();

    // Spawn a game execution thread, start it up, and stash it in session.
    // Also be sure to give it a WebContext so Comet will work from it,
    // as well as any other information it will need to run.
    WebContext webContext = WebContextFactory.get();
    Thread opponent = new Opponent();
    ((Opponent)opponent).setWebContext(webContext);
    ((Opponent)opponent).setTileValues(opponentGrid);
    opponent.setDaemon(true);
    opponent.start();
    HttpSession session = inRequest.getSession(true);
    session.setAttribute("opponent", opponent);

    // Also store in session the number of matches the player has found
    // during this game. Used to check for a win.
    session.setAttribute("matches", new Integer(0));

    log.trace("startGame() - Exit");
    return userGrid;

  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }

} // End startGame().
```

The first thing we see is a call to a method, `generateGrid()`. That's next on our agenda, but for now it's enough to understand that its job is to return an array of values for a grid, each value defining which image is under each tile.

After that there is some code that's definitely interesting. You see, the way things work here is that your computerized opponent is actually a background thread running on the server. The thread, which is an instance of the `Opponent` class, gets stashed in the user's session, so it's unique to each player playing the game. It's set up as a daemon thread, which is necessary in most containers to ensure it can be shut down when the container goes down and won't gum up the works, so to speak. Two other things the thread needs are a `WebContext` instance and, of course, the array of tile values generated earlier.

But it's that `WebContext` that's of most interest to us. As you'll recall from earlier, this object is essentially what allows us to be "connected" to the client. But, this object actually requires the context of a request. So, think this through for a second . . . a call is made from the browser to the server to start the game. This is a request. When this method completes, that request is done. So, how can this thread possibly send information back to the server?

The answer is that long-running request that I mentioned you can see in Firebug. As it turns out, the `WebContext` you get here knows about that request (this is all under-the-covers kind of stuff by the way, and while it's good to understand it, it's enough to just know it works).

So, by keeping a reference to the `WebContext` in the thread, that thread essentially has access to that long-running request, so it can send information back to the browser without a new request coming in. Precisely what we want!

Aside from the `Opponent` thread, we also put a simple attribute named `matches` in session and set its value to an `Integer` with the value zero. This records how many matches the player has found and is used to determine whether the player has won or not.

Once that's done, the grid of tile values is returned so that it can be used on the client, which we've already seen.

Let's now go back to the `generateGrid()` method I mentioned a second ago. Here is that method's code:

```
private int[] generateGrid() {

  log.trace("generateGrid() - Entry");

  // The grid of tiles.
  int[] grid = new int[42];

  // Record which tiles have been randomly set already.
  HashMap<Integer, Object> tilesSetAlready = new HashMap<Integer, Object>();

  // There are 21 tile types to set, i.e., 21 pairs.
  for (int i = 1; i < 22; i++) {
    Random generator = new Random();
    // Pick one tile, and keep doing it until one that isn't set is found.
    int tile1 = generator.nextInt(42);
    while (tilesSetAlready.get(new Integer(tile1)) != null) {
      tile1 = generator.nextInt(42);
    }
    // Pick another tile, and keep doing it until one that isn't set is found.
    tilesSetAlready.put(new Integer(tile1), new Object());
    int tile2 = generator.nextInt(42);
    while (tilesSetAlready.get(new Integer(tile2)) != null) {
      tile2 = generator.nextInt(42);
    }
    tilesSetAlready.put(new Integer(tile2), new Object());
    // Set the tiles to the next tile type.
    grid[tile1] = i;
    grid[tile2] = i;
  }

  log.trace("generateGrid() - Exit");
  return grid;

} // End generateGrid().
```

Since the goal is to place two copies of each of the 21 different tile types (because, remember, there are 42 tiles in a grid), we begin a loop from 1 to 21. For each, we randomly choose a

tile number from 0 to 41. Then, we see whether that tile has already been chosen by looking it up in the `tilesSetAlready Map`. If it's already been used, we pick another tile and try again. If it's not used, we record it in that `Map` and select another, and again do the same check and repeat the choice if it's already used. Finally, when we have two tiles that haven't been previously set, we go ahead and set their values to the current value of the loop, and then continue the loop. At the end, we return the resultant array.

The next method found in the `GameCore` class is the `checkForWin()` method, and it's what's called each time the player finds a set of matching tiles.

```
@RemoteMethod
public boolean checkForWin(HttpSession inSession) {

  // Bump up the player match count.
  int matches = ((Integer)inSession.getAttribute("matches")).intValue();
  matches++;
  inSession.setAttribute("matches", new Integer(matches));

  // See if the player won.
  if (matches > 20) {
    // Yes, player won, so stop opponent thread.
    ((Opponent)inSession.getAttribute("opponent")).setIsActive(false);
    return true;
  } else {
    // Nope, player didn't win.
    return false;
  }

} // End checkForWin().
```

Now we can see how that `matches` session attribute is used. Each time a match is found, that value is bumped up. When it hits `21`, the player has won, and `true` is returned; otherwise, `false` is returned. It's just that simple!

And speaking of simple, only one method remains in this class, and that's `howToPlay()`, which is that method I mentioned earlier that uses DWR's ability to read from other URLs. Now we can see what that's really all about:

```
@RemoteMethod
public String howToPlay() throws Exception {

  return WebContextFactory.get().forwardToString("/howToPlay.txt");

} // End howToPlay.
```

The long and short of this is that using the `forwardToString()` method of the `WebContext` class (which is what you get out of the call to `WebContextFactory.get()` naturally) does the equivalent of a standard forward from a servlet. The difference is that instead of immediately returning the result of that forward to the client, it's returned as a string. So here, we're executing a JSP and getting the result of that execution, and then returning it from this method. Since it's a forward, the path you specify as the argument to `forwardToString()` must begin

with a slash because it's context-relative. This is a really handy feature because what it allows you to do is continue to use the templating technology of your choice as the mechanism that generates output to return to the caller of a remote method. This saves you from having to write a lot of string manipulation code in your classes, which is definitely a cleaner way to do things.

OpponentCreator.java

Now, before we actually look at this `OpponentCreator` class, let's discuss why it's necessary in the first place.

You will recall from looking at the client code that there are some calls to the `Opponent` object. So, we need DWR to be able to create an instance of the `Opponent` class. However, in looking at `GameCore`'s `startGame()` method, you saw that an instance of `Opponent` is created and stashed in session. As you'd expect, we want to be getting *that* instance and executing methods on it when we make remote calls to `Opponent`. So, how can we make DWR do that?

The answer is that DWR by default can't do it, thus we need to write our own `Creator` to handle it.

Writing a custom `Creator` is a simple matter of implementing the `Creator` interface. Additionally, you may want to extend the `AbstractCreator` class, which fills in some of the more boilerplate-type code you'd otherwise have to deal with. I've done that here, and the result is that only two methods really need to be implemented.

In Figure 8-7, you can see the simple UML diagram for `OpponentCreator`.

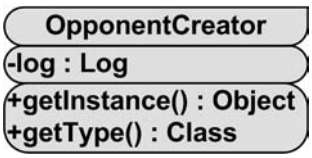


Figure 8-7. *UML class diagram of the* `OpponentCreator` *class*

In Listing 8-4, the source for the entire class is shown.

Listing 8-4. *The* `OpponentCreator` *Class in Its Entirety*

```
package com.apress.dwrprojects.inmemoria;


import javax.servlet.http.HttpSession;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.directwebremoting.create.AbstractCreator;
import org.directwebremoting.extend.Creator;
import org.directwebremoting.WebContext;
import org.directwebremoting.WebContextFactory;
```

```
/**
 * A custom DWR creator that retrieves the Opponent object from the caller's
 * session and returns it.
 *
 * @author <a href="mailto:fzammetti@omnytex.com">Frank W. Zammetti</a>
 */
public class OpponentCreator extends AbstractCreator implements Creator {


  /**
   * Log instance.
   */
  private static Log log = LogFactory.getLog(OpponentCreator.class);


  /**
   * This method gets the Opponent object from the session associated with
   * the caller and returns it.
   *
   * @return The Opponent instance.
   */
  public Object getInstance() throws InstantiationException {

    log.trace("getInstance() - Entry");
    WebContext webContext = WebContextFactory.get();
    HttpSession session = webContext.getSession();
    Opponent opponent = (Opponent)session.getAttribute("opponent");
    log.trace("getInstance() - Exit");
    return opponent;

  } // End getInstance().


  /**
   * Return a Class for the type of object getInstance() returns.
   *
   * @return A Class instance.
   */
  public Class getType() {

    log.trace("getType() - Entry");
    Class clazz = null;
    try {
      clazz = Class.forName("com.apress.dwrprojects.inmemoria.Opponent");
    } catch (Exception e) {
      // Shouldn't ever happen.
      e.printStackTrace();
    }
```

```
    log.trace("getType() - Exit");
    return clazz;

  } // End getType().


} // End class.
```

The two methods we care about are `getInstance()` and `getType()`. The `getInstance()` method is what DWR calls to get an instance of the class being remoted. In this case, that means we need to get a handle to the user's session, which we can get via the `WebContext` object. Then we simply pull the `Opponent` instance from session and return it.

The `getType()` method is used by DWR to return the appropriate type of object. Note that the return type from `getInstance()` is `Object`. While I haven't examined the code in DWR, I can take a guess what's happening: DWR is using this information to automagically cast that `Object` to the appropriate type and then call the requested method. The bottom line though is that these two methods are required to make everything work, and isn't that what's really important anyway?!?

Opponent.java

Only one server-side class remains to be examined, and it's the most lengthy and most complex by far, and that's the `Opponent` class. We've of course seen this class a number of times already, but now it's time to dig in and see what makes it tick, beginning with its UML diagram in Figure 8-8.

Figure 8-8. *UML class diagram of the* `Opponent` *class*

Let's begin examining the actual code by looking into the class fields present in the class. In Table 8-2, you can see the summary of the fields and what they are.

Table 8-2. *The Fields Found in the* `Opponent` *Class*

Name	Description
`log`	The usual Commons Logging `Log` instance seen in most of the server-side code throughout this book.
`isActive`	Flag: Is this opponent still active (meaning the game hasn't ended yet)?
`webContext`	A DWR `WebContext` instance for the session representing the human player.
`tileValues`	The grid of tiles generated when the game was started.
`tileStates`	The grid of tile states. `0` means the tile is available, `1` means it's been matched already.
`flippedTile1`	The field that records the first tile flipped by the opponent.
`flippedTile2`	The field that records the second tile flipped by the opponent.
`matches`	The number of matches the opponent has found thus far.
`STATE_WAITING`	A constant describing the thread when in the wait state.
`STATE_PICK_TILE_1`	A constant describing the thread when in the state where it is picking a first tile.
`STATE_PICK_TILE_2`	A constant describing the thread when in the state where it is picking a second tile.
`STATE_CHECK_FOR_MATCH`	A constant describing the thread when in the state where it is checking to see whether the two chosen tiles match.
`currentState`	The field that stores what state the opponent is currently in.

The next step is taking a look at the annotation on the class itself, which is this:

```
@RemoteProxy(creator=OpponentCreator.class)
```

Ah, now we can see in practice what we discussed earlier with regard to the attributes available on the `@RemoteProxy` annotation. Here, we're telling DWR to use the `OpponentCreator` class we just looked at to create instances of this class when methods are remotely called on it. Now the picture is complete!

Now we move on to the first method in this class. Well, it isn't **quite** the first method in the class because there are actually two setter methods, one for the `webContext` field and one for the `tileValues` field. Both of these are called when the `Opponent` is set up in `GameCore`, hence the need for the setters.

Moving on, note that this class extends `Thread`, and as such it has a `run()` method, like any good Java `Thread` class does. Oh, and look, here's that method now:

```
public void run() {

  while (isActive) {

    try {
```

```
log.trace("run() - Next iteration");

ScriptBuffer script = new ScriptBuffer();

switch (currentState) {

  case STATE_PICK_TILE_1: {

    log.trace("run() - Picking tile #1");
    flippedTile1 = pickTile(false);
    currentState = STATE_WAITING;
    script.appendScript("inMemoria.opponentFlipTile(").appendData(
      "1").appendScript(", ").appendData(flippedTile1).appendScript(
      ", ").appendData(tileValues[flippedTile1]).appendScript(");");

  break; }

  case STATE_PICK_TILE_2: {

    log.trace("run() - Picking tile #2");
    flippedTile2 = pickTile(true);
    currentState = STATE_WAITING;
    script.appendScript("inMemoria.opponentFlipTile(").appendData(
      "2").appendScript(", ").appendData(flippedTile2).appendScript(
      ", ").appendData(tileValues[flippedTile2]).appendScript(");");

  break; }

  case STATE_CHECK_FOR_MATCH: {

    log.trace("run() - Checking for match");
    if (tileValues[flippedTile1] == tileValues[flippedTile2]) {
      // Match.
      log.debug("run() - Match found");
      tileStates[flippedTile1] = 1;
      tileStates[flippedTile2] = 1;
      script.appendScript("inMemoria.opponentMatch(").appendData(
        flippedTile1).appendScript(", ").appendData(
        flippedTile2).appendScript(");");
    } else {
      // No match.
      log.debug("run() - No match");
      script.appendScript("inMemoria.opponentNoMatch(").appendData(
        flippedTile1).appendScript(", ").appendData(
        flippedTile2).appendScript(");");
    }
    currentState = STATE_WAITING;
```

```
        break; }

      }

      // Call client-side functions.
      ScriptSession scriptSession = webContext.getScriptSession();
      scriptSession.addScript(script);

      sleep(250);

    } catch (Exception e) {
      // This shouldn't ever happen.
      e.printStackTrace();
    }

  } // End isActive loop.

} // End run().
```

This is a pretty typical Java thread method, so no surprises in its basic structure. First things first, we get a `ScriptBuffer` instance because regardless of what happens next (an exception notwithstanding), we're going to be sending something to the client.

Next we have a `switch` on `currentState`. The value of `currentState` determines what this iteration of the thread should do. There are essentially three things the thread could do: pick a first tile, pick a second tile, or see whether the two chosen tiles match. There is a fourth state, denoted by the `STATE_WAITING` constant, but that means exactly what it would seem to mean: the thread is waiting for something, specifically acknowledgment from the client that the last command the thread sent to it has completed, so any iteration that executes during that state will result in nothing happening.

So, what happens when the thread picks the first tile (and actually, the same thing happens when picking the second tile too)? The first thing that happens is a call to `pickTile()`, which returns a number from `0` to `42`. Next, the thread is put in the waiting state. Finally, that `ScriptBuffer` we got earlier is used to call the `InMemoria.opponentFlipTile()` method, passing it the number of the tile being flipped (either `1` or `2`), the tile number that was chosen, and what image appears there. This will cause the client to flip the tile and then send an acknowledgement back to the thread. Recall when we looked at the client-side `InMemoria` class that there were four methods I was going to leave out until later, because it made more sense to look at them in the context of the server-side code? Well, the time has arrived to do so! Here is the `opponentFlipTile()` method:

```
this.opponentFlipTile = function(inSelectionNumber, inTileIndex,
  inImageIndex) {

  inTileIndex++;
  dwr.util.byId("tileOpponent_" + inTileIndex).src =
    this.tileImages[inImageIndex].src;
  Opponent.confirmFlip(inSelectionNumber);

} // End opponentFlipTile().
```

So, what's happening here? First, the incoming tile index is incremented by one. This is because the number chosen is zero-based, while the numbering of the images in the grid is one-based. Next, the `src` of the tile is changed to the `src` of the specified image from the `tileImages` array. Finally, the `confirmFlip()` method of the `Opponent` class (this is the server-side `Opponent` class, remember) is called. The `confirmFlip()` method is this code:

```
@RemoteMethod
public void confirmFlip(final int inTile,
  final HttpSession inSession) {

  log.trace("confirmFlip() - Entry");
  if (log.isDebugEnabled()) {
    log.trace("confirmFlip() - inTile = " + inTile);
  }
  if (inTile == 1) {
    currentState = STATE_PICK_TILE_2;
  } else {
    currentState = STATE_CHECK_FOR_MATCH;
  }
  inSession.setAttribute("opponent", this);
  log.trace("confirmFlip() - Exit");

} // End confirmFlip().
```

First, you can see here the `@RemoteMethod` annotation to tell DWR that this method is callable from the client. This is one of two methods that is so annotated, the other being `outcomeComplete()`, which we'll see later.

This method is responsible for updating the state of the thread. If the first tile was just chosen (and flipped by the client), which is indicated by the argument to the method, the state is set so that the thread will pick a second tile on the next iteration. If it was instead the second tile that was chosen and flipped, the next iteration needs to check whether they match. Either way, the `Opponent` is updated in session.

THERE'S MORE THAN ONE WAY TO SKIN A CAT!

During the course of writing this chapter, I had the opportunity to meet Joe Walker at The Ajax Experience in Boston, and wound up cohosting a session there with him. During the "rehearsal" for that session (if you can call sitting around in the morning for a few minutes eating bagels, drinking coffee, and looking at some slides together rehearsal!), he turned me on to something I was not aware of: there is more than one way to call methods on the client. For instance, instead of the string concatenations you see here that are added to a `ScriptBuffer`, you can instead do code like this:

```
Collection<ScriptSession> sessions =
  serverContext.getScriptSessionsByPage("index.jsp");
ScriptProxy proxy = new ScriptProxy(sessions);
proxy.addFunctionCall("inMemoria.opponentNoMatch",
  new Integer(flippedTile1), new Integer(flippedTile2));
```

What this says is "Give me all the users "connected" to the `index.jsp` page (meaning they have a reverse Ajax connection open), and then call the `inMemoria.opponentNoMatch()` on each of them." Now, of course, we only have a single user in this application, but the concept works there too.

This code is a little cleaner and more concise in my opinion, but I think it's pretty close to six of one and a half-dozen of the other. In any case, the application was already written when I learned about this approach, so the application wasn't changed. And now you've seen the alternative too, and actually learned something besides, because this method allows you to send something to all connected users, which is, of course, a very useful thing to be able to do with reverse Ajax.

When it's time to check for a match, the third case of the switch in the `run()` method executes. If they match, the values in the `tileStates` array is updated to "1" to indicate the tiles have already been matched. Then, the `opponentMatch()` method of `InMemoria` is called on the client, which is this method:

```
this.opponentMatch = function(inTile1, inTile2) {

  inTile1++;
  inTile2++;
  inMemoria.opponentFlippedTile1 = inTile1;
  inMemoria.opponentFlippedTile2 = inTile2;
  inMemoria.opponentExplosionFrame = 0;
  inMemoria.opponentExplosionDir = true;
  inMemoria.opponentExplosionInterval =
    setInterval("inMemoria.cycleOpponentExplosion()", 100);

} // End opponentMatch().
```

First, the two tile values are incremented to account for the zero-based vs. one-based array difference between the client and server. Next, `opponentFlippedTile1` and `opponentFlippedTile2` are set to the values passed in. This is needed by the `cycleOpponentExplosion()` method, which is then set to fire at an interval, just like we saw done for the player. Once the animation cycle is complete, the `outcomeComplete()` method is called, which is this code:

```
@RemoteMethod
public void outcomeComplete(final HttpSession inSession) {

  log.trace("outcomeComplete() - Entry");
  if (tileValues[flippedTile1] == tileValues[flippedTile2]) {
    // Match. Bump up match counter and check for a win.
    matches++;
    if (matches > 20) {
      log.debug("outcomeConfirmer() - Opponent won");
      // Opponent won. Stop thread from running and alert the client.
      isActive = false;
      ScriptBuffer script = new ScriptBuffer();
```

```
      script.appendScript("inMemoria.opponentWon();");
      ScriptSession scriptSession = webContext.getScriptSession();
      scriptSession.addScript(script);
      currentState = STATE_WAITING;
      inSession.setAttribute("opponent", this);
      log.trace("outcomeComplete() - Exit (1)");
      return;
    }
  }
  // Reset for next tile pick. Note this won't matter if the opponent won.
  flippedTile1 = -1;
  flippedTile2 = -1;
  currentState = STATE_PICK_TILE_1;
  inSession.setAttribute("opponent", this);
  log.trace("outcomeComplete() - Exit (2)");

} // End outcomeComplete().
```

This is, as you can see, the other remotable method in this class; all the others are private and therefore not callable from the client. This is a pretty simple method as well. First, simply compare the values of the two flipped tiles. If they match, bump up the value of the `matches` field. If that value is now greater than `20`, the opponent has won, in which case `isActive` is set to `false` so that no further iterations occur. Then, again using the `ScriptBuffer` object, we write out a call to the `opponentWon()` method of the `InMemoria` class. Update the `Opponent` object in session, and we're all done.

Note that when the tiles don't match, things are substantially the same, except that it's the `opponentNoMatch()` method of `InMemoria` that is called. The only real difference between that and `opponentMatch()` is that none of the explosion stuff is necessary; all that's really needed is a call to `unFlip()` to get the tiles back into their unflipped state. At the end, `outcomeComplete()` of the `Opponent` object is again called.

Now, back on the client, the `opponentWon()` method gets executed, which is shown here:

```
this.opponentWon = function() {

  inMemoria.playerProcessClicks = false;
  alert("Sorry, your opponent has defeated you.  Try again, puny human!");

} // End opponentWon().
```

No rocket science here, that's for sure! Setting `playerProcessClicks` to `false` ensures the human player cannot click any more tiles, and the game is then over with an oh-so-pretty `alert()` message (and I wouldn't be me if I didn't taunt the loser a bit!).

Now, if it turns out that the two tiles did **not** match, the two fields `flippedTile1` and `flippedTile2` are set to `-1` to indicate they have not yet been chosen, and the thread is put in the state to pick another first tile.

Speaking of picking a tile, we skipped over the `pickTile()` method earlier, so let's take a look at that now:

```
private int pickTile(final boolean inSecondTile) {

  Random generator = new Random();
  if (inSecondTile) {
    // When it's the second tile, there's initially a 10% chance the opponent
    // will simply choose the right matching tile; otherwise fall through to
    // a random choice. However, as the opponent finds more matches, the
    // percentage increases slightly to simulate a player with a memory
    // (albeit not a great memory!).
    int choosePercent = generator.nextInt(100);
    if (choosePercent < (10 + matches)) {
      log.debug("pickTile() - Choosing matching tile");
      for (int i = 0; i < 42; i++) {
        if (tileValues[i] == tileValues[flippedTile1] && i != flippedTile1) {
          if (log.isDebugEnabled()) {
            log.debug("pickTile() - Matching tile = " + i);
          }
          return i;
        }
      }
    }
  }
  int tile = generator.nextInt(42);
  // Keep picking until we pick one that isn't already matched and that
  // isn't currently flipped.
  while (tileStates[tile] == 1 || tile == flippedTile1 ||
    tile == flippedTile2) {
    tile = generator.nextInt(42);
  }
  if (log.isDebugEnabled()) {
    log.debug("pickTile() - tile = " + tile);
  }
  return tile;

} // End pickTile().
```

This should be relatively straightforward code, but the one thing that bears some explanation is the game "logic." As it turns out, if you have the opponent simply picking tiles at random, the opponent will very nearly always lose because the player is, of course, intelligent (presumably!), and this will beat random more times than not. So, I needed to introduce a little bit of advantage to the computer opponent. The way I did this is that some percentage of the time, the opponent will just magically choose the correct matching tile, if it's the second

tile being chosen. Much more than 10 percent, and the opponent is just too good and will beat the human purely almost every time (it was just trial and error to find the right value here). I also increase that percentage with each match found, which kind of simulates a real player who tends to remember tiles as he or she plays longer. Again though, incrementing the percentage more than one percentage point per match tends to make the opponent too "smart" and too difficult to beat.

So, now that we've seen the code for the flow between the client and server and back again, I thought it would be helpful to see this in a graphic form, and Figure 8-9 is just such a form.

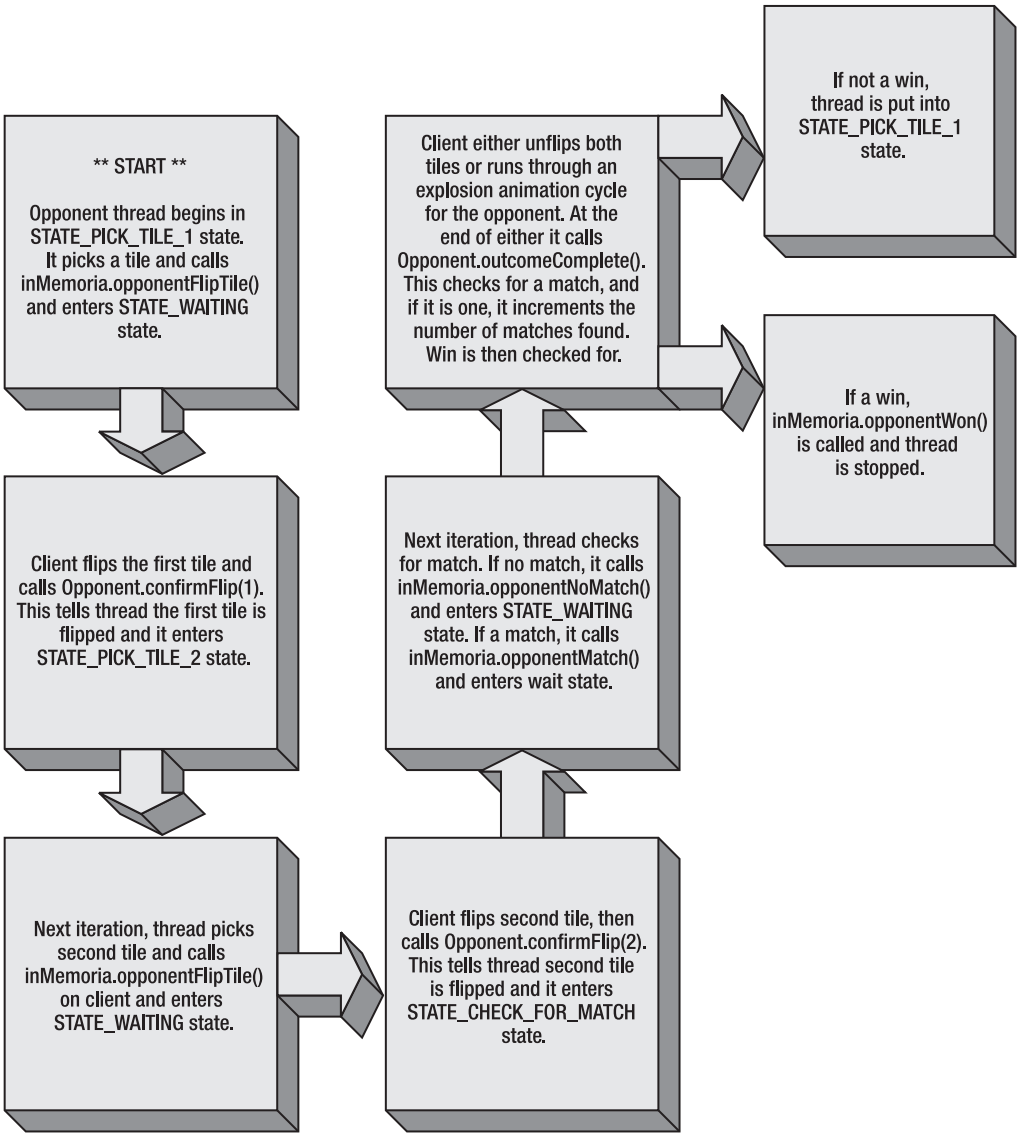

** START **
Opponent thread begins in STATE_PICK_TILE_1 state. It picks a tile and calls inMemoria.opponentFlipTile() and enters STATE_WAITING state.
Client flips the first tile and calls Opponent.confirmFlip(1). This tells thread the first tile is flipped and it enters STATE_PICK_TILE_2 state.
Next iteration, thread picks second tile and calls inMemoria.opponentFlipTile() on client and enters STATE_WAITING state.
Client flips second tile, then calls Opponent.confirmFlip(2). This tells thread second tile is flipped and it enters STATE_CHECK_FOR_MATCH state.
Next iteration, thread checks for match. If no match, it calls inMemoria.opponentNoMatch() and enters STATE_WAITING state. If a match, it calls inMemoria.opponentMatch() and enters wait state.
Client either unflips both tiles or runs through an explosion animation cycle for the opponent. At the end of either it calls Opponent.outcomeComplete(). This checks for a match, and if it is one, it increments the number of matches found. Win is then checked for.
If not a win, thread is put into STATE_PICK_TILE_1 state.
If a win, inMemoria.opponentWon() is called and thread is stopped.

Figure 8-9. *Flow chart showing the opponent move cycle*

I hope that when you examine this diagram and then look back at the code a little bit, the whole picture of how the server uses reverse Ajax to call the client, and then the client acknowledges that some requested action has taken place, all becomes clear. Don't feel bad if it seems a little confusing—it certainly is! Besides that, the server calling code on the client is far outside what most developers are used to, so you shouldn't feel remotely bad about having to take a second look to really internalize it (and if you just understood it completely the first time through, well, you're smarter than me, that's for sure!).

Suggested Exercises

This project is a little harder to come up with good exercises for, frankly, because most of them involve making the game better, and that's not going to give you a whole lot of DWR practice. Still, I've given it a shot:

- Introduce a couple of different difficulty levels. Allow users to choose what difficulty they want when they start a game, and send a request to the server to set it on the `Opponent` object. Hint: the difficulty can be easily changed by adjusting that "magic percentage" and the amount it increases with each match.
- Allow for a multiplayer mode where two human players can play against each other.
- Try to use some of the other DWR-supported reverse Ajax methods and see what they change. Heck, just see whether the game is still viable under other methods!

Summary

Well, after the very large previous chapter, it sure was nice to come up against a far shorter and simpler one. It seemed like the perfect time for a little break, and a game is a great way to take a break. But, we touched on a number of interesting things here nonetheless. First, we saw annotation-based configuration of DWR for the first time. We then saw reverse Ajax in action and how background processes go hand-in-hand with it pretty naturally in many cases. We also got to write our own custom `Creator`, and we learned how we can read from other URLs and return it as the output from a remote call.

Only one chapter to go now, and there we'll use many of the skills we've learned throughout this book and of course be introduced to a few new ones. Get ready, the ride is nearing the end, but there's still plenty to learn!

CHAPTER 9

Timekeeper: DWR Even Makes Project Management Fun!

Like Frodo, Odysseus, and John Sheridan, we've traveled a great journey, but all great journeys must come to an end, and the end of ours is near. Here, in the final chapter of this little adventure that we've shared together, we have one final application to examine, a few final things to learn, and a last little bit of fun to have.

In this chapter, we'll be building a project management/timekeeping application. To do this, we'll employ DWR, naturally, but we'll also employ some new tools, namely Hibernate and HSQLDB for our data access layer and Ext JS for our UI widget needs. We'll put all these pieces in the hopper and watch as a useful application emerges from the other end, and in the process we'll be exposed to a few more tricks up DWR's sleeve, not to mention see some great supporting libraries up close and personal.

This is it, the final leg of our journey that began in Chapter 1 and now concludes in Chapter 9. Let's step into the abyss and see what lies beyond the rim!

Application Requirements and Goals

The application in this chapter is entitled Timekeeper, and its purpose, in brief, is to allow us to track time booked against various projects that workers are assigned to. It will also alert us to when projects are running over their allotted time, are on time for their target completion, or when there appears to be more time allocated than necessary so that resources can be reallocated more efficiently if desired. For users assigned to projects, Timekeeper is all about timesheets.

Let's face it: a timesheet application isn't all that mind-boggling in either complexity or the fun factor. However, we can make it more interesting for both users as well as us software developers by mixing in some Web 2.0 goodness.

So, let's now enumerate the features and requirements we intend to implement:

- Users of Timekeeper can be administrators, which means they can create projects and users.
- Users of Timekeeper can be project managers, which means they can assign other users to projects and change the parameters of projects.

- Clearly, the ability to create, edit, and delete both users and projects will be necessary. We won't go overboard in terms of how much information is stored about each, but we'll have enough to make the application useful.
- Every user should be greeted with a "dashboard"-type screen upon logging in that gives an overview of the projects of interest to them. On this same screen should be a shortcut to their current timesheet so they can book time against projects they are working on right away. In addition, this dashboard should update in real time whenever any user makes a change to the data that is displayed on it (if this sounds like a job for reverse Ajax, you're right!).
- The UI for the application should be kind of cool looking because after all, most users don't particularly like timesheets and booking time, so let's make it as pretty to look at as we can.
- We're going to need a database, and while we've seen Derby in action in a couple of other projects and know it would do the trick just fine, let's play with a new toy this time around if we can find one (and, of course, you know we can!).
- From a technical standpoint, the projects in this book that have had database requirements have had JDBC code in them, and while it was fairly well abstracted out from the core application code, let's see if we can't avoid that altogether by using a very popular data access layer tool called Hibernate.

So, that's what we're trying to accomplish here. Two of the requirements, that of the data access layer and the desire to make the UI a bit flashy, will be provided by two supporting libraries, namely Hibernate and Ext JS. The database itself will be provided by another open source package named HSQLDB, and that seems like a good place to start.

HSQLDB

HSQLDB, formerly called Hypersonic SQL, is a lightweight, 100 percent pure Java SQL database engine. It supports a number of modes, including in-memory (for use with applets and such), embedded (which is how we'll be using it here), and client-server mode, which is basically a stand-alone database server. HSQLDB is the embedded database engine used in OpenOffice, and that is a pretty good pedigree if you ask me!

You can find HSQLDB's web site here: `www.hsqldb.org`. One of the great things about HSQLDB is how drop-dead simple it is to use. First, one of its very convenient features is that if you attempt to access a database that does not exist, it will go ahead and create it for you. So, to get a database set up, this is all we have to do:

```
Class.forName(Globals.getDbDriver()).newInstance();
Connection conn =
  DriverManager.getConnection("jdbc:hsqldb:c:\temp\myDatabase",
  "sa", "");
conn.close();
```

This will create a new directory named `myDatabase` in `c:\temp` and will create a basic database for us. At this point, we can go ahead and use standard JDBC and SQL to create tables, insert data, or whatever we want to do. There is no complex startup procedure and no setup code. There are not even any classes to import specific to HSQLDB! As it turns out, Hibernate will effectively do this for us under the covers, so we don't even need to do **that** much work, but I wanted to show you how easy it is to get started even if you weren't going to use Hibernate, as we are in this project.

HSQLDB is housed in a single JAR file with no outside dependencies. The JAR file is less than 1MB in size, so when they say "lightweight," they aren't kidding! Yet, it supports many features, such as views, temp tables and sequences, referential integrity, triggers, transaction support, Java stored procedures and functions, and even `Object` data types!

When you need to have database functions in your application, and if you do not have or want a full-blown relational database management system (RDBMS), HSQLDB is a great solution. In fact, even if you do have a full-blown RDBMS like Oracle or SQL Server, you might want to think about using HSQLDB anyway.

Hibernate

If you are familiar with Hibernate, by all means, feel perfectly free to skip this section, as this is intended to be just a brief introduction to those readers who may not be familiar with it yet. I suspect 99 percent of the people reading this book will in fact skip this section, but if you're in that 1 percent, don't feel bad one bit, and continue reading!

Hibernate (`www.hibernate.org`) is what's called a *professional open source project*. This means that while the project is available for free, as is the full source, just like any other open source project, its development is financially supported by some commercial entity, which almost always offers some sort of value-add services, such as support for enterprises looking to use the project in their efforts. In the case of Hibernate, that commercial entity is JBoss (`www.jboss.com`), a division of Red Hat (`www.redhat.com`), they of Linux fame. The commercial offering of JBoss is 100 percent optional; there are absolutely **zero** requirements that you pay one red cent to use Hibernate. However, if you use it in an enterprise setting, you may well **want** to pay for the added benefits you get, including a higher level of technical support.

In any case, Hibernate is what's known as an Object-Relational Mapping (ORM) tool. An ORM tool provides a way of mapping relational data stores, a typical relational database for instance, to object-oriented representations in a typical OOP language, such as Java. With ORM tools, you create Java (or .NET, in the case of NHibernate, a port of Hibernate to the .NET Framework) classes, and Hibernate takes care of the database for you. You don't think in terms of tables and other typical database concepts, but just classes.

Hibernate also provides a portable SQL-like language called HQL, short for Hibernate Query Language, that allows you to query and get back objects (or collections of objects) just like you would with SQL. This eliminates the differences in SQL dialects that you sometimes encounter across different database vendors' platforms.

Hibernate truly deals with **all** of the database concerns you'd typically have to take on yourself. In fact, Hibernate can create the database for you on the fly! This is in fact precisely what will happen with Timekeeper. You won't find any of the table creation code you saw in previous projects; Hibernate will take care of all of that for us.

Hibernate also deals with automatic primary key creation for the objects you persist to the database. Hibernate allows you to have associations of various kinds between objects, so you lose none of the power you've likely grown accustomed to in the realm of straight JDBC programming.

I'm going to skip showing you any code examples here as I've usually done in previous projects. I believe Hibernate is a lot easier to understand when viewed in context, so that's precisely what we'll do as we dissect the application together.

TO ORM OR NOT TO ORM, THAT IS THE QUESTION!

There are some pretty strong feelings on both sides of the question of whether ORM tools are something that should be embraced or not. There are good arguments on both sides, but thankfully it's your choice to make!

My own personal opinion, for what it's worth, is how I typically approach any technology decision: analyze the particulars of a given situation and choose the best tool for the job (a lesson my father taught me all those years ago when I tried to put a nail in a piece of wood with a large pair of pliers just because it's what I had handy!).

There's something to be said for getting away from all the frequently tedious JDBC coding that Hibernate saves you from. There's something to be said for approaching everything from a purely object-oriented standpoint and architecting your applications from that standpoint alone. There's something to be said for only having to have one set of knowledge and expertise available.

Then again, there's something to be said for having more control of both the code that accesses your database and the database itself. There's something to be said for using an RDBMS to its full potential and not losing some of the power by abstracting it away. There's something to be said for removing layers of complexity that may or may not be as beneficial as you might think.

In this application, we're letting Hibernate truly do it all, from creating the database schema to coming up with the SQL statements to execute. You can, however, exercise more control and still use Hibernate. You can create the database yourself and make Hibernate conform to it. You can hand-write the SQL that Hibernate will use. It's not an either-or proposition. But, I would argue that to do those things defeats a big benefit of Hibernate, and so you have to decide whether it's important enough to you or not.

There are trade-offs, without a doubt, and I don't believe there is a one-size-fits-all answer. Some feel that we should always use Hibernate, or some other ORM tool, and there should be no debate. I for one do not agree with that and believe it leads to bad architectures in at least some cases. No, I believe my father was right: if you need a hammer, go get it out of the toolbox. If it's a screw and not a nail though, the hammer will still probably work, but it's not the right tool for the job!

Ext JS

With Hibernate on board to make our database requirements a piece of cake, let's discuss how we'll make the UI easier and fancier. The answer there is a popular JavaScript library called Ext JS (`http://extjs.com`). While Ext JS does more than just widgets, that is frankly what it's primarily known for, and for very good reason!

Ext JS provides a wealth of UI widgets that cover a wide field of needs, including

- Grids, both static and editable, and all sorts of variations of that theme
- Various forms of pop-up windows, both purely informational and data entry types
- Tabbed panes and other kinds of layout-related components
- Entry form–type controls such as combo boxes and autocomplete fields
- Toolbars, menus, progress bars, and tons more

All of these widgets look really, really good, and what's more are fully skinnable so you can have any sort of look you can dream up (and write style sheets for). Ext JS also offers various types of effects in most of these widgets so that it's not just some pretty pictures but a toolkit you can use to create dynamic, active interfaces.

In addition to the widget goodness Ext JS gives you, it also contains a whole cast of utility functions such as

- Functions to determine the type of browser
- The typical functions used to get references to page elements in various ways
- Some utilities for working with strings, numbers, and dates (the date-related functions are quite far-reaching actually)
- Functions to help working with arrays
- Some extensions to JavaScript functions themselves that provide additional methods on every function object

These truly only scratch the surface of what Ext JS offers, especially with regard to widgets.

In addition, Ext JS takes a very object-oriented approach to **everything**, which leads to a clean, logical interface. It all just makes sense as you write code using Ext JS.

As with Hibernate, I will refrain from going into usage details here because really there isn't any generic code I could show that wouldn't be redundant when we look at the code of the application. Instead, I'll discuss aspects of Ext JS as they are actually used in the application code.

And speaking of the application code, it's about that time we start tearing it apart and seeing what makes it tick. Before we do that though, let's get a first glimpse of Timekeeper, as shown in Figure 9-1.

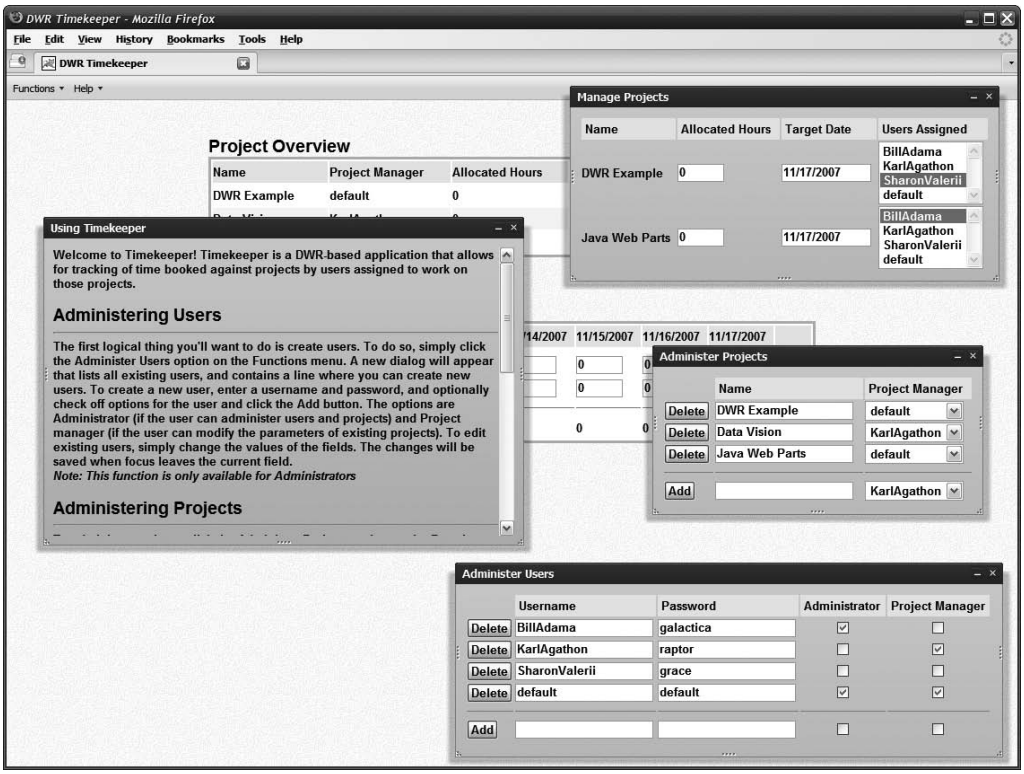

DWR Timekeeper - Mozilla Firefox
File Edit View History Bookmarks Tools Help
DWR Timekeeper
Functions Help
Project Overview
Name Project Manager Allocated Hours
DWR Example default 0
Manage Projects
Name Allocated Hours Target Date Users Assigned
DWR Example 0 11/17/2007
Java Web Parts 0 11/17/2007
BillAdama
KarlAgathon
SharonValerii
default
Using Timekeeper
Welcome to Timekeeper! Timekeeper is a DWR-based application that allows for tracking of time booked against projects by users assigned to work on those projects.
Administering Users
The first logical thing you'll want to do is create users. To do so, simply click the Administer Users option on the Functions menu. A new dialog will appear that lists all existing users, and contains a line where you can create new users. To create a new user, enter a username and password, and optionally check off options for the user and click the Add button. The options are Administrator (if the user can administer users and projects) and Project manager (if the user can modify the parameters of existing projects). To edit existing users, simply change the values of the fields. The changes will be saved when focus leaves the current field.
Note: This function is only available for Administrators
Administering Projects
11/15/2007 11/16/2007 11/17/2007
Administer Projects
Name Project Manager
Delete DWR Example default
Delete Data Vision KarlAgathon
Delete Java Web Parts default
Add KarlAgathon
Administer Users
Username Password Administrator Project Manager
Delete BillAdama galactica
Delete KarlAgathon raptor
Delete SharonValerii grace
Delete default default
Add

Figure 9-1. *Timekeeper, a first glimpse*

This also serves as a good demonstration of some of what Ext JS has to offer, including some pop-ups and the menu bar. Another example, which is what users first see after logging in (assuming they are assigned to projects), can be seen in Figure 9-2.

So, now that we have an idea what the application looks like, let's start the show and get to some code!

The Ext JS web site has a large number of live demos you can look at that I suggest taking a few minutes here to peruse. Something else you will find there is their API documentation, which, rather than just being some static HTML pages, is actually a full-fledged Ext JS–based application itself! In fact, if you look at nothing else, I'd suggest browsing that documentation because you'll see a good bit of what makes Ext JS cool there alone!

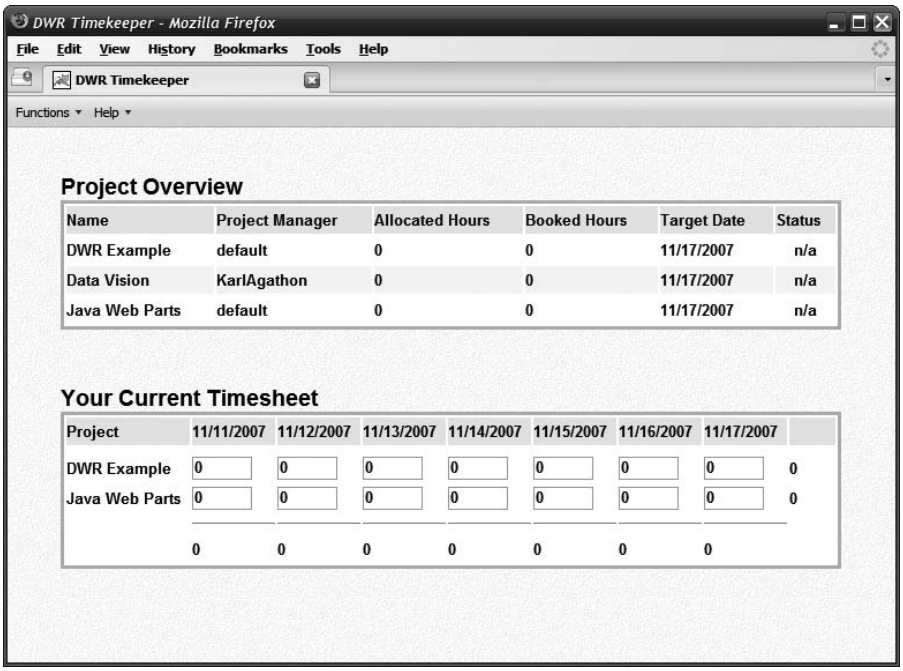

DWR Timekeeper - Mozilla Firefox
File Edit View History Bookmarks Tools Help
DWR Timekeeper
Functions Help
Project Overview
Name Project Manager Allocated Hours Booked Hours Target Date Status
DWR Example default 0 0 11/17/2007 n/a
Data Vision KarlAgathon 0 0 11/17/2007 n/a
Java Web Parts default 0 0 11/17/2007 n/a
Your Current Timesheet
Project 11/11/2007 11/12/2007 11/13/2007 11/14/2007 11/15/2007 11/16/2007 11/17/2007
DWR Example 0 0 0 0 0 0 0 0
Java Web Parts 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0

Figure 9-2. *The Timekeeper home page, after logging in*

Dissecting Timekeeper

By this point, you should be rather tired of hearing me (so to speak!) say that this application's directory structure is a typical Java webapp structure, but umm, that's exactly what it is! Figure 9-3 shows the structure, and although it's nothing unusual, there are a few new things to see, and we'll go over it all just the same because I want you to get your money's worth after all!

In the root directory, we find a single `index.jsp` file that is all the markup of the application and the starting point. Interestingly, it contains both the login screen and the main application screen itself, plus all the pop-ups, but I'm getting ahead of myself.

The first directory is the usual `css` directory, which contains the usual `styles.css` file containing the style sheet for the entire application, nice and externalized, which is just how we like it!

The `img` directory contains all the graphics for the application. There aren't actually too many: `background.gif` is the textured background of the main application screen; `borderline.gif`, `late.gif`, and `ok.gif` are all the status icons that visually tell the current status of a project; `login0.gif` and `login1.gif` are the normal and hover states for the login screen's login button; `logon_bottom.gif` and `logon_left.gif` form the pretty and somehow tangentially relevant image on the logon page; `spacer.gif` is a single-pixel transparent GIF used for, what else, spacing purposes!

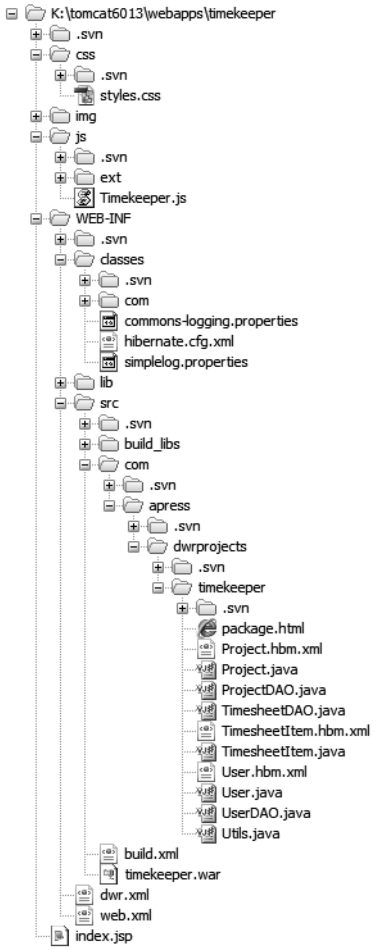


Figure 9-3. *The directory structure of the application*

Into the `js` directory we go next, and here we find something old and something new. The something old is a single JavaScript file named `Timekeeper.js` that houses all the client-side code for the application. The something new is a subdirectory named `ext`. This contains all the Ext JS files, JavaScript source files, as well as image files, style sheets, and any other resources the library may need.

Under `WEB-INF` we find all the typical contents including `web.xml`, `dwr.xml`, the `lib` directory with library JARs, and the `src` directory containing all the source code for the application. In the `classes` directory, we find the Commons Logging configuration files as in all the other projects in this book, and we also find something else new, namely the `hibernate.cfg.xml` file, which is the Hibernate configuration file.

Speaking of configuration files, that's our next stop on the train of learning (why does that sound like something from *The Great Space Coaster*?[1]), so let's get off there and have a look around, shall we?

One final note before we move forward: throughout this chapter, I've condensed some of the source code in an effort to get the size of this chapter down just a little bit. Mostly this amounts to removing stand-alone comment lines and removing some whitespace between blocks of code. The actual source you should have downloaded by now retains all that.

Configuration Files

As far as configuration files go, we have two that we've seen plenty of times by now, and three new ones.

web.xml

The `web.xml` file for this application is shown in Listing 9-1, although there isn't anything new to see in it.

Listing 9-1. *The* `web.xml` *File for Timekeeper*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app id="timekeeper" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>TimeKeeper</display-name>
  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>crossDomainSessionSecurity</param-name>
      <param-value>false</param-value>
    </init-param>
```

---

1. *The Great Space Coaster* (`http://en.wikipedia.org/wiki/The_Great_Space_Coaster`) was a children's television show from the early '80s that many of us in our mid 30s grew up with. Most people tend to remember two things: Knock-Knock the bird, who naturally enough told knock-knock jokes, and Gary Gnu, who did the fake news reports ("No gnews is good gnews with Gary Gnu"). Of course, the theme song tends to stick in our heads too: ". . . get onboard, step inside, slowly for a magic ride . . . roaring towards the other side where only rainbows hide . . ." Ah, the memories!

```
    <init-param>
      <param-name>activeReverseAjaxEnabled</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
       <param-name>classes</param-name>
       <param-value> com.apress.dwrprojects.timekeeper.UserDAO,
         com.apress.dwrprojects.timekeeper.ProjectDAO,
         com.apress.dwrprojects.timekeeper.TimesheetDAO
       </param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>30</session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Since we'll be using annotation-based DWR configuration in this application, you can see the classes that will be configured in that way listed as the value of the `classes` parameter. You can also see that reverse Ajax is enabled via the `activeReverseAjaxEnabled` parameter, since we'll be using that capability of DWR as well. Other than those items, this is not really any different from what we've previously looked at, so we won't dwell on it any longer.

dwr.xml

Err, wait, didn't I just say right before this that we'd be using annotation-based configuration to tell DWR about our remotable classes? Indeed I did, so why exactly would a `dwr.xml` file be present here? The answer is simple: DWR allows you to mix and match XML-based configuration and annotation-based configuration at your discretion, and that's exactly what I've done with this application. So, the annotations in the classes will work in conjunction with the configuration information present in the `dwr.xml` file shown in Listing 9-2 seamlessly.

In the case of conflicting annotations and XML configuration, the XML configuration prevails. For example, if you have a method in a class annotated with `@RemoteMethod`, and you then specify that method in `dwr.xml` as excluded via an `<exclude>` tag, the method will not be remotable.

Listing 9-2. *The* `dwr.xml` *Configuration File*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">
<dwr>
  <allow>
    <convert converter="bean"
      match="com.apress.dwrprojects.timekeeper.User" />
    <convert converter="bean"
      match="com.apress.dwrprojects.timekeeper.Project" />
    <convert converter="bean"
      match="com.apress.dwrprojects.timekeeper.TimesheetItem" />
  </allow>
</dwr>
```

The way I've chosen to split the configuration between the configuration file and annotations is to put all the converter specifications in the configuration file and all the create specifications in annotations. Remember though, there are no rules around this; you are free to delineate configuration in any way you see fit, in any combination.

Here we simply have three Java classes that DWR will be allowed to marshal back and forth to JavaScript, and all three use the built-in bean converter. The contents of each class will be discussed later, but they are nonetheless pretty obvious I think!

hibernate.cfg.xml

The `hibernate.cfg.xml` file gives Hibernate the basic information it will need to connect to the database underlying the application. The contents of this file are shown in Listing 9-3.

Listing 9-3. *The* `hibernate.cfg.xml` *Configuration File*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:file:timekeeper</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.pool_size">3</property>
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
    <property name="current_session_context_class">thread</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>
    <mapping resource="com/apress/dwrprojects/timekeeper/Project.hbm.xml"/>
    <mapping resource="com/apress/dwrprojects/timekeeper/User.hbm.xml"/>
```

```
    <mapping resource="com/apress/dwrprojects/timekeeper/TimesheetItem.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```

While the ins and outs of configuring Hibernate are largely out of the scope of this book, it's helpful to go over the basics. The first four elements, those with names beginning with `connection`, mimic the JDBC connection parameters you'd need to connect to any database via JDBC.

Following that is configuration of connection pooling via the `connection.pool_size` property, which Hibernate can provide for you (or it can be provided by another library and Hibernate can make use of it). Here we are in fact using Hibernate's built-in connection pooling, and we're telling Hibernate to allow for three connections in this pool. This can then be adjusted as necessary for your user load.

While Hibernate effectively abstracts the details of dealing with any particular database from us, it still under the covers has to know how to talk to that database; and since most databases implement their own peculiar dialect of SQL, Hibernate needs to know which dialect to use, and that's precisely what the `dialect` property tells it. Hibernate supports most dialects you're likely to run into out of the box, including HSQLDB, which is convenient since that's our underlying data store in this application!

The `current_session_context_class` property has to do with how Hibernate will deal with sessions. In a nutshell, setting this value to `thread` as is done here tells Hibernate that the session is scoped to the current Java thread executing a transaction.

Setting the `show_sql` property to `true`, as the next property does, just causes Hibernate to display all executed SQL statements to stdout. This is good for debugging purposes and seeing what Hibernate is doing under the covers.

The `hbm2ddl.auto` property is next and it is, for my money, the most interesting property of the bunch. This tells Hibernate that when it starts up, it should update the database schema with the configurations taken from the various `.hbm` files (don't worry; we look at what they're all about next). In short, if you make a change to the objects you're dealing with, this property tells Hibernate to update the underlying schema as well automatically. You can also set this to `create-drop`, among other settings, which would cause the schema to be dropped and recreated fresh on every startup, which can be helpful during development, but obviously not usually such a good idea in a production application!

I'll be the first to admit I'm not a big-time Hibernate user, and because of this I may not have described all of this in as much depth as you may like, especially if Hibernate is new to you. In that case, check out `www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html`, which goes into greater detail on all of this than I do (or can) go into here.

Those `.hbm` files I mentioned are up next, and in short, they tell Hibernate what the underlying database schema should look like, but in the context of Java objects.

Project.hbm.xml

For every Java class you wish to use in the context of Hibernate, you need to tell Hibernate a little bit about it. This is done in the form of `.hbm` files. The name of these files is in the form `xxx.hbm.xml`, where `xxx` is the class name, and their location is the same place you would put the `.class` file it describes (you can in fact put them elsewhere, but this **is** typical, and, in my opinion at least, makes a lot of sense—refer to the Hibernate docs if you'd like to put them somewhere else).

One such file is `Project.hbm.xml`, which describes the `Project` class for use with Hibernate. This file is shown in Listing 9-4.

Listing 9-4. *The Hibernate Definition File for the* `Project` *Class,* `Project.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.apress.dwrprojects.timekeeper.Project" table="PROJECTS">
    <id name="id" column="PROJECTS_ID">
      <generator class="native" />
    </id>
    <property name="name" />
    <property name="projectManager" />
    <property name="allocatedHours" />
    <property name="bookedHours" />
    <property name="targetDate" type="timestamp" />
    <property name="usersAssigned" />
  </class>
</hibernate-mapping>
```

I'm going to go out on a limb here and guess you've already surmised that the `Project` class describes a project known to Timekeeper. This file then tells Hibernate how it relates to a database. As you can see, we specify the name of the class, and then tell Hibernate the name of the database table that an object of this class will be stored in.

The `<id>` element tells Hibernate about the unique value that identifies a given object, which all Hibernate-managed objects need. Here, we are saying that there should be a column in the `PROJECTS` table named `PROJECTS_ID` that will uniquely identify a given `Project`, and that when a `Project` is saved to the database Hibernate should use the native ID generator to assign a value to this column. This generator is one that comes with Hibernate, and it's a simple number incremented by one with each object saved.

Following this is a series of `<property>` elements. These simply tell Hibernate the names of the fields in the `Project` class that will be persisted to the database. By default, the column names will match the names shown here, although you can override them if you prefer (since the idea of Hibernate is to keep you from having to deal with such details, I prefer to specify as little as possible in my `.hbm` files). Note the `targetDate` field, which specified the type as `timestamp`. This is because sometimes a Java type doesn't have an exact match to a database

type, and Hibernate needs a bit of a hint as to what type to use, as is the case oftentimes with date-type data types.

I suspect this file, even if you are completely new to Hibernate, isn't mind-blowing by any stretch. It is, as are the other two `.hbm` files, largely self-explanatory and not very difficult to grasp.

User.hbm.xml

In many ways, if you've seen one `.hbm` file, you've seen them all, and that's really quite true in this application where they are pretty simplistic. But, that won't stop us from looking at the next one, shown in Listing 9-5, which describes the `User` class for Hibernate.

Listing 9-5. *The Hibernate Definition File for the* `User` *Class,* `User.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.apress.dwrprojects.timekeeper.User" table="USERS">
    <id name="id" column="USERS_ID">
      <generator class="native" />
    </id>
    <property name="username" />
    <property name="password" />
    <property name="isAdministrator" />
    <property name="isProjectManager" />
  </class>
</hibernate-mapping>
```

This looks very much like the file for the `Project` class, even a little simpler! We won't spend any more time on it frankly, as there's nothing new to see in this one.

TimesheetItem.hbm.xml

The final Hibernate config file to look at is for the `TimesheetItem` class, which is used to describe a single entry in a user's timesheet, and it is shown in Listing 9-6.

Listing 9-6. *The Hibernate Definition File for the* `TimesheetItem` *Class,* `TimesheetItem.hbm.xml`

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.apress.dwrprojects.timekeeper.TimesheetItem"
    table="TIMESHEETS">
    <id name="id" column="TIMESHEETS_ID">
      <generator class="native" />
```

```
    </id>
    <property name="userID" />
    <property name="projectID" />
    <property name="reportDate" />
    <property name="hours" />
  </class>
</hibernate-mapping>
```

As was the case for the `User` config file, it's almost boring at this point, no? You can even easily figure out what each field is for, in this file and the previous two, but I'll list them out nonetheless later when we look at these classes themselves. For now, I'd say we're about done with configuration files.

Now it's time to get into some actual code, and as has become habit throughout the preceding chapters of this book, we begin by looking at the client-side code that makes up the Timekeeper application.

The Client-Side Code

Only a few files make up the client-side code of Timekeeper, in fact, the exact number is three! Although the number of files is small, they have some decent meat to them, so let's jump right in.

styles.css

All the style sheet information in Timekeeper is nicely externalized into the style sheet file named `styles.css`, found in the `css` directory, and shown in Listing 9-7.

Listing 9-7. *The* `styles.css` *File*

```
/* Catch-all style. */
* {
  font-family      : arial;
  font-size        : 10pt;
  font-weight      : bold;
}

/* Style for body. */
.cssBody {
  margin           : 0px;
  padding          : 0px;
  background-image : url("../img/background.gif");
}

/* Style for outer container div. */
.cssOuter {
  margin           : 4px;
  padding          : 4px;
}
```

```
/* Style for source of dialog animations. */
.cssSource {
  position         : absolute;
  left             : 1px;
  top              : 1px;
  width            : 1px;
  height           : 1px;
}

/* Style for heading on home page. */
.cssHeader {
  font-size        : 14pt;
}

/* Style of a table on the home page. */
.cssTable {
  background-color : #ffffff;
}

/* Style of container of tables on home page. */
.cssHomeTableContainer {
  border           : 3px solid #98c0f4;
}

/* Style for table headers. */
.cssTableHeader {
  background-color : #dfe8f6;
}

/* Style for regular table row. */
.cssTableRow {
  height           : 24px;
}

/* Style for alternate table row. */
.cssTableAltRow {
  height           : 24px;
  background-color : #e0ffe0;
}

/* Styles for Ext widgets. */
#toolbar {
  margin-left      : 20px;
  width            : 320px;
}
#menubutton {
  margin-left      : 20px;
```

```
}
.msg .x-box-mc {
  font-size        : 14px;
}
#msg-div {
  position         : absolute;
  left             : 35%;
  top              : 0px;
  width            : 250px;
  z-index          : 20000;
}
```

This style sheet begins, almost ritualistically at this point, with the catch-all "wildcard" style that ensures all text on the page will have a consistent style.

Next is the `cssBody` class, which sets some styles on the `<body>` element. Most importantly is removal of all margin and padding so that the menu bar sits flush against the top of the content area, and the background image that gives a little bit of a texture to the background of the home page.

The `cssOuter` class is used to style the `<div>` that the home page sits in, and its purpose is to provide some padding around the edges of the content; that is, we don't want the content of the home page bumping up against the menu bar, or the left and right edges of the content area, and this class does that for us.

The `cssSource` class is interesting . . . whenever the dialog boxes for things like editing projects and users are shown, it's done so with a nice little animation. This animation needs a starting point, a source location in the window. For the sake of simplicity, I decided that the upper-left corner of the window would do the trick nicely. This style is applied to a `<div>` that serves as that source, since Ext JS requires the source be an element on the page.

The `cssHeader` class styles the larger text headings on the home page.

The `cssTable` class styles the two tables on the home page. Its only purpose in life is to ensure the background of the table is white and thus stands out from the textured background a bit.

The `cssTableHeader` class is, as you've guessed I'm sure, the style applied to the headings of the tables, and its job is to give a bluish background color to that row of the table to make the headers stand out, as headers are wont to do!

The `cssTableRow` and `cssTableAltRow` styles are applied in an alternating fashion to the row of the tables to give a nice striping effect that helps guide the user's eyes across the data and keep things straight as he or she reads.

After that is a group of four styles that pertain to Ext JS widgets. The first two determine sizing characteristics of the menu bar, and the final two deal with the slide-down message that appears periodically when updates are done to the server. In all honesty, these last two I copied from one of the Ext JS demos, just altering the sizing a bit to make the message a little harder to miss.

index.jsp

The `index.jsp` file contains the markup of Timekeeper, every last bit of it, and interestingly it effectively contains two "screens," if you will. The first is the logon page that you are greeted

with when you first access the application, and the second is the main screen where you actually interact with the application.

The `index.jsp` file is somewhat long, so I'll be showing portions of it here rather than the entire thing in one shot, and the first piece to look at is this:

```
<%@ page language="java" import="com.apress.dwrprojects.timekeeper.*" %>

<html>
  <head>
    <title>DWR Timekeeper</title>
    <link rel="stylesheet" type="text/css" href="css/styles.css">
    <script src="dwr/engine.js"></script>
    <script src="dwr/util.js"></script>
    <script src="dwr/interface/UserDAO.js"></script>
    <script>
      dwr.engine.setErrorHandler(errorHandler);
      function errorHandler(inMessage) {
        alert(inMessage);
      }
    </script>
```

This chunk of code always renders and forms the beginning of the response to the client. As you can see, all the classes in the `timekeeper` package are imported. This is because in a short while we'll need access to the `User` class. After that is a pretty typical document `<head>` section, including imports of various DWR JavaScript files. You may realize at this point that not all the proxy objects are imported, and that's because this piece of the JSP will always execute, but after this comes a branch that will either render the logon screen or the main screen, and in each of those cases we'll need different proxy files. I didn't see any reason to import JavaScript files that weren't needed for one branch or the other. Lastly, you can see we set up a simple, generic error handler for any exceptions thrown during DWR calls.

Following this section is the first of two branches, as shown here:

```
<%
  /* ---------- If user isn't logged in, show login. ---------- */
  if (session == null || session.getAttribute("user") == null) {
%>
    <script>
      function doLogin() {
        UserDAO.logUserIn(
          dwr.util.getValue("username"),
          dwr.util.getValue("password"),
          { callback : function(inResp) {
              if (inResp == "Ok") {
                window.location = "index.jsp";
              } else {
                alert("Invalid credentials.  Please try again.");
```

```
            }
          }
        }
      );
    }
  </script>
</head>
<body>
  <table border="0" width="100%" cellspacing="0" cellpadding="0">
    <tr>
      <td rowspan="2" width="305">
        <img src="img/logon_left.gif"></td>
      <td height="295" valign="middle">
        Please log in to begin:
        <br><br>
        <table border="0" cellpadding="2" cellspacing="2">
          <tr>
            <td class="cssTableHeader">Username </td>
            <td><input type="text" name="username" value=""></td>
          </tr>
          <tr>
            <td class="cssTableHeader">Password </td>
            <td><input type="text" name="password" value="default"></td>
          </tr>
          <tr>
            <td> </td>
            <td align="right">
              <img src="img/login0.gif" onClick="doLogin();"
                onMouseOver=
"this.style.cursor='pointer';this.src='img/login1.gif';"
                onMouseOut="this.src='img/login0.gif';">
            </td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td height="178" valign="bottom">
        <img src="img/logon_bottom.gif">
      </td>
    </tr>
  </table>
```

This code is responsible for the logon screen, as shown in Figure 9-4.

Figure 9-4. *The logon page of the application*

As you can see, the branch condition is whether an attribute under the name `user` is found in session (and if a session has been established). If a session has not been established, or if the attribute is not found in session, the logon screen is needed because the user is not yet logged in.

So, the JSP continues to execute and output content, starting with a `<script>` block that contains a DWR call to the `logUserIn()` method of the `UserDAO` class on the server. Two arguments are passed to this method, those being the username and password entered by the user. If a simple "OK" response comes back, the user logs in, which is a simple matter of reloading this same JSP because the `else` portion of this branch renders the main screen, as we'll see shortly. If any other response comes back, an `alert()` is shown indicating the user was not logged in and he or she should try again.

The actual HTML that follows this `<script>` block is actually relatively unremarkable. About the only semi-interesting thing is the way the large graphic and title are put together, along with the form the user fills out. A table that is actually a 2×2 grid is used. The large graphic was split in half, but when put into table cells, assuming padding and margins and alignment have been done properly on the table (which I'm happy to say is the case!), it looks like a giant, seamless image, with a form in a location that might be a little tricky to accomplish without this splitting trick. Admittedly, it's not much of a trick per se, web designers have been doing this for ages, and in far more complex forms usually; but still, it's an effective little technique that can allow you to achieve layouts that might otherwise be difficult or even

impossible to accomplish. (Think things like circular layouts, for example: if you carve a larger image up into multiple pieces and reassemble it properly in a table, or CSS-based layout if you prefer that approach, you can do nonrectangular designs that you couldn't otherwise do.)

As I said, the `else` portion of this branch is executed when the user has successfully been validated. Here's how the code for that branch begins (but this is **not** the full code of the branch, as that's a much larger chunk):

```
<%
  /* ---------- If user IS logged in, show the app. ---------- */
  } else {
    User user = (User)session.getAttribute("user");
%>

    <script type="text/javascript"
      src="js/ext/adapter/yui/yui-utilities.js"></script>
    <script type="text/javascript"
      src="js/ext/adapter/yui/ext-yui-adapter.js"></script>
    <script type="text/javascript" src="js/ext/ext-all.js"></script>
    <link rel="stylesheet" type="text/css"
      href="js/ext/resources/css/ext-all.css">

    <script src="dwr/interface/ProjectDAO.js"></script>
    <script src="dwr/interface/TimesheetDAO.js"></script>

    <script src="js/Timekeeper.js"></script>

  </head>

  <body class="cssBody" onLoad="timekeeper.init(<%=user.getId()%>);">
```

The first chore in this branch is to go fetch the `User` object that is placed in session during a successful login (by the `logUserIn()` method of the `UserDAO` class, as we'll see down the road a bit). Once that's done, we can continue building up the response, and hence the HTML document the browser will render, beginning with a series of resource imports, both JavaScript and style sheets, for Ext JS. After that are imports of the other DWR proxy JavaScript files that weren't needed for the logon page but are needed for the main application. Finally, we have an import of the `Timekeeper.js` file, which contains all the actual client-side code of the application.

In the `<body>` tag, you'll also notice an `onLoad` event handler that calls the `init()` method of the instance of the JavaScript `Timekeeper` class, and more interestingly is what's passed to this method. Remember, we're still looking at a JSP, so we can grab the `id` field of the `User` object pulled from session, which is a numeric value, and pass it as an argument to this method. This is a piece of information we need to keep handy on the client, and doing this allows the `Timekeeper` class to do so.

Following that chunk of code is a whole lot of plain old HTML. Because of its size, I'm going to refrain from listing whole sections of it here and instead ask that you have a read through of it off to the side before continuing. I promise: it really is just straight HTML!

You'll find a couple of discrete sections as you read through it. The first you encounter is the markup for the home page itself. After that comes five sections that are interesting because they are used by Ext JS as the contents of the various dialog boxes that can be shown, for instance, administering users and projects. The way it works is that you create the markup you want to appear in the dialog box, and then tell Ext JS to display the dialog box, pointing it at your content, and it takes care of turning your content into an actual dialog. But, at the end of the day, you're just writing straight HTML, nothing special.

As an example, in Figure 9-5 is the Manage Projects dialog box.

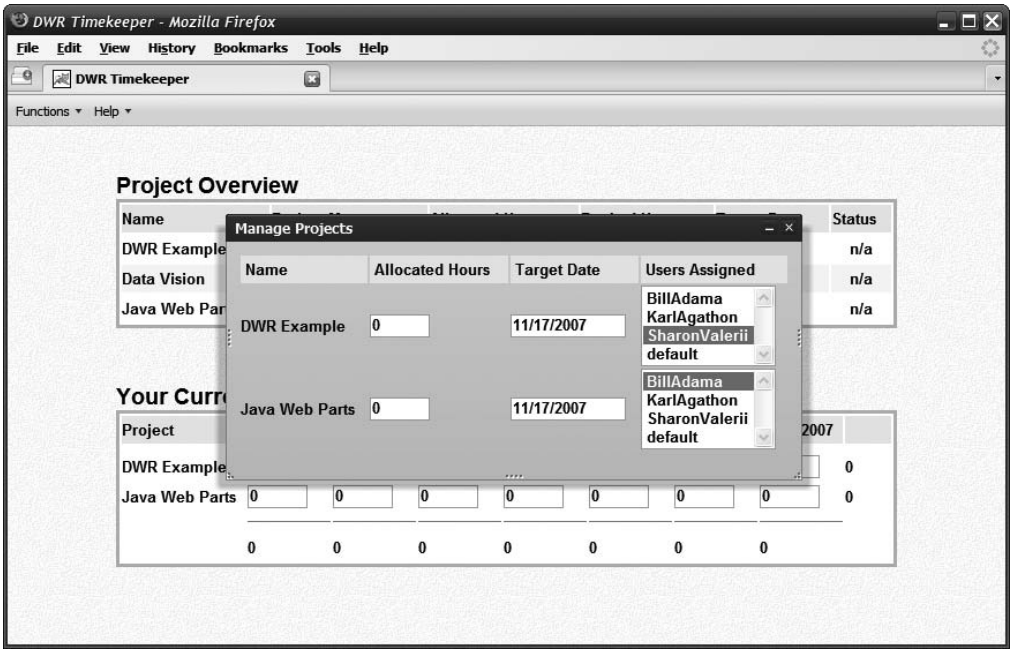


Figure 9-5. *The Manage Projects dialog box*

The corresponding markup in `index.jsp` for this dialog box is this:

```
<div id="dialogManageProjects"
  style="visibility:hidden;position:absolute;top:0px;">
  <div class="x-dlg-hd">Manage Projects</div>
  <div class="x-dlg-bd">
    <table border="0" width="100%">
      <thead>
        <tr style="height:24px;">
          <td class="cssTableHeader"> Name </td>
          <td class="cssTableHeader"> Allocated Hours </td>
          <td class="cssTableHeader"> Target Date </td>
          <td class="cssTableHeader"> Users Assigned </td>
        </tr>
      </thead>
```

```
      <tbody id="divManageProjects_projectList"></tbody>
    </table>
  </div>
</div>
```

See, I really wasn't lying; it's just good ole HTML, nothing more! As I said, have a look at the rest just to get an overview of the structure of the document, but even if you don't do that, the preceding example is likely more than sufficient to give you enough of the picture to be able to move forward without a deeper look. The only sections that are even remotely different are the markup for the Using Timekeeper dialog box, as shown in Figure 9-6.

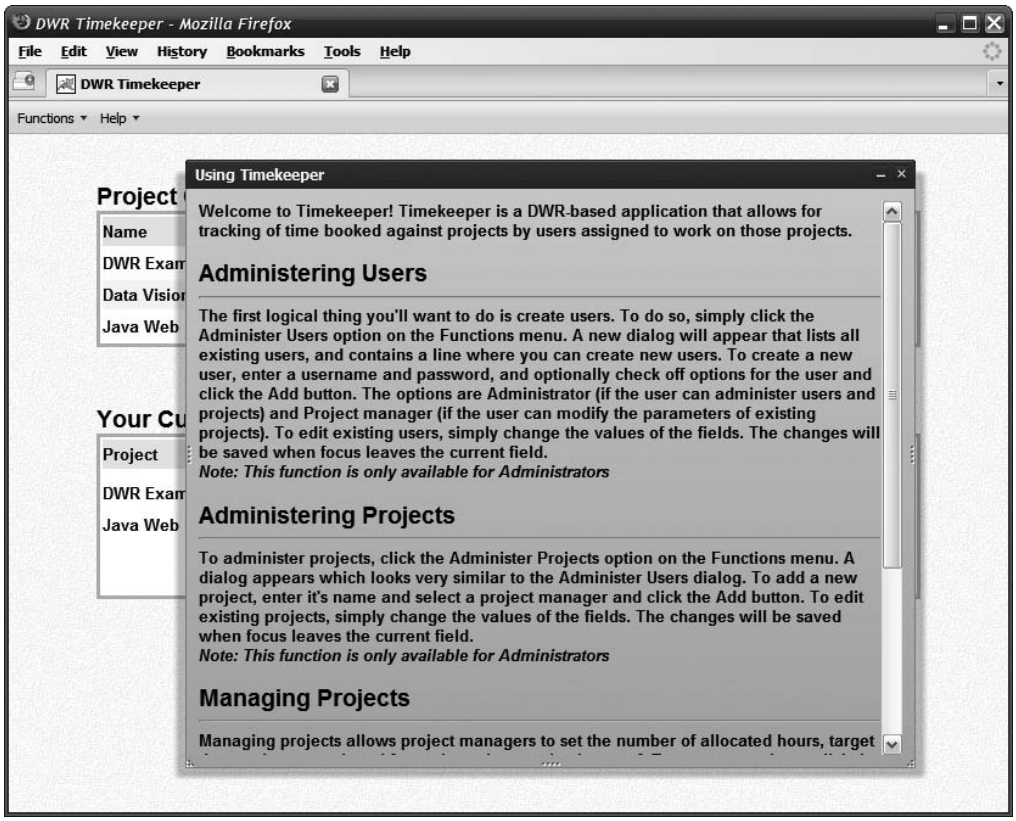

DWR Timekeeper - Mozilla Firefox
File Edit View History Bookmarks Tools Help
DWR Timekeeper
Functions Help
Using Timekeeper
Welcome to Timekeeper! Timekeeper is a DWR-based application that allows for tracking of time booked against projects by users assigned to work on those projects.
Administering Users
The first logical thing you'll want to do is create users. To do so, simply click the Administer Users option on the Functions menu. A new dialog will appear that lists all existing users, and contains a line where you can create new users. To create a new user, enter a username and password, and optionally check off options for the user and click the Add button. The options are Administrator (if the user can administer users and projects) and Project manager (if the user can modify the parameters of existing projects). To edit existing users, simply change the values of the fields. The changes will be saved when focus leaves the current field.
Note: This function is only available for Administrators
Administering Projects
To administer projects, click the Administer Projects option on the Functions menu. A dialog appears which looks very similar to the Administer Users dialog. To add a new project, enter it's name and select a project manager and click the Add button. To edit existing projects, simply change the values of the fields. The changes will be saved when focus leaves the current field.
Note: This function is only available for Administrators
Managing Projects
Managing projects allows project managers to set the number of allocated hours, target
Project
Name
DWR Exam
Data Visior
Java Web
Your Cu
Project
DWR Exam
Java Web

Figure 9-6. *The Using Timekeeper dialog box*

And the only real difference is that these are actually **simpler** sections!

Timekeeper.js

With the style sheet and markup out of the way, we're ready to move into the meat of the client-side code, which is the JavaScript contained in the `Timekeeper.js` file, and correspondingly, the `Timekeeper` JavaScript class.

Weighing in at a hair over 1,000 lines of code, this clearly isn't something that should be dumped onto the pages of a book in its entirety, so let's dissect it piece by piece, breaking where it makes sense. You'll find that there is one rather large and complex method, while the rest aren't nearly as daunting.

The Data Fields

The `Timekeeper` class contains a number of data fields, which are summarized in Table 9-1.

Table 9-1. *The Data Fields in the* `Timekeeper` *Class*

Field Name	Description
`dialogs`	An object that stores handles to the various dialog boxes.
`currentUser`	The ID of the user logged in. This is where the ID passed into the `init()` method comes into play.
`users`	A list of `User` objects representing all users known to Timekeeper.
`projects`	A list of `Project` objects representing all projects known to Timekeeper.
`timesheetItems`	A list of `TimesheetItem` objects for all the time the current user has booked.
`oneDay`	The number of milliseconds in one day. This is used when calculating whether a project is on time, overdue, or borderline.
`msgCt`	The count of message boxes present. This is used to allow for multiple slide-down messages if requests are made to the server back to back in rapid succession.
`functionsMenu`	A reference to the functions menu. This is needed to manipulate the menu later.

Miscellaneous Methods

I've grouped the methods according to the general function they perform, or more accurately, what they operate on, be it users, projects, or something else. After that grouping is done, a few methods naturally fall outside any of those scopes, and those methods are listed in this miscellaneous section.

initUI() In many of the previous applications, you saw an `init()` method, and in fact, there's one here as well. So the natural question to ask, aside from why there have been four Guitar Hero games and yet not a single Dream Theater or Queensryche[2] track in any of them, is why there's an `initUI()` method too? The answer is that Ext JS provides a handy event hook:

2. Guitar Hero, in case you've been living in Bora Bora for a few years, is a game for various game consoles where you, what else, play guitar! It comes with an actual plastic guitar controller that hooks up to your console and it has a number of buttons on the neck and a strum switch on the body. Scrolling colored circles flow down the screen and you need to hit these on the guitar controller in rhythm with the music. I mention Dream Theater and Queensryche, two popular progressive rock bands, because they have never appeared in any of the four existing Guitar Hero bands. This is a bit crazy because John Petrucci from Dream Theater is acknowledged as one of the top guitar players alive today, possibly even the best, and Queensryche is known for impressive lead/rhythm guitar parts mixed together (the Guitar Hero games allow you to have multiple guitar controllers hooked up so one player can play lead and another can play rhythm or even bass parts).

```
Ext.onReady(timekeeper.initUI, timekeeper, true);
```

You'll find this line at the very end of `Timekeeper.js`, so it executes when that file loads. The `onReady` event differs from the usual `onLoad` event (which is where `init()` is called from) in that `onReady` fires as soon as the DOM is fully loaded and ready for manipulation, but potentially **before** all the content for the page has loaded, and perhaps more importantly, **before** anything is rendered on the screen. This is ideal because oftentimes when you're building a UI programmatically, as we're doing with Ext JS, if you used `onLoad`, you'd see the UI being built little by little, with elements perhaps being replaced little by little, which just plain looks bad. If you do all that stuff before rendering anything to the screen though, you can avoid all that ugliness. So, we have an `initUI()` method, which builds the user interface, and this occurs before anything is rendered on the screen, so for the user the UI simply appears, as it should, with no intermediate steps.

So, what's actually going on in `initUI()`? Lots of Ext JS stuff, of course!

```
this.initUI = function() {

Ext.QuickTips.init();

timekeeper.functionsMenu = new Ext.menu.Menu({
  id : "functionsMenu",
  items: [
       { id : "functionsAdministerUsers", text : "Administer Users",
         disabled : true, handler : timekeeper.menuClickHandler },
       { id : "functionsAdministerProjects", text : "Administer Projects",
         disabled : true, handler : timekeeper.menuClickHandler },
       new Ext.menu.Separator({}),
       { id : "functionsManageProjects", text : "Manage Projects",
         disabled : true, handler : timekeeper.menuClickHandler }
  ]
});
```

First, we tell Ext JS to initialize quick tips, which is another word for tooltips. Note that some default tooltips are created by Ext JS, using the text as their content. Then, we begin by building the menu bar. A menu bar in Ext JS is a toolbar with drop-down menu items on it (that's actually only **one** way to implement a menu bar with Ext JS—feel free to explore their docs to see other ways). So, we first create a Functions menu. The items on the menu are, I think, self-explanatory. The handler element is simply the function that will be called when the item is clicked. I've routed all menu item clicks through the same method of the `timekeeper` object to keep the code consolidated.

Also, take note that all three menu items are by default disabled. This will come into play shortly.

The next step is to build the menu bar itself:

```
var tb = new Ext.Toolbar("divMenubar", [
  {
    text : "Functions", menu : timekeeper.functionsMenu
  },
  {
```

```
    text : "Help",
    menu : {
      id : "mnuHelp",
      items : [
        { text : "Using Timekeeper", handler : timekeeper.menuClickHandler },
          new Ext.menu.Separator({}),
        { text : "About Timekeeper", handler : timekeeper.menuClickHandler }
      ]
    }
  }
]); // End menu creation.
```

As I said previously, the menu bar is actually a `Toolbar` object. For each item, we attach a `Menu` instance. In the case of the Functions menu, it's the `Menu` object we created previously. For the Help menu, it's actually created inline. I just did it two different ways to show you it's possible. You can also see a `Separator` instance used in both menus, which is just a dividing line to separate menu items. The menu bar is attached to the element on the page with the ID `divMenubar`, which is the first argument passed to the Toolbar constructor.

With the menu constructed, it's now time to turn our attention to creating the pop-up dialog boxes for all the functions the application provides. Doing so couldn't be simpler! Here's building the Administer Users dialog box:

```
if (timekeeper.dialogs["dialogAdministerUsers"] == null) {
  timekeeper.dialogs["dialogAdministerUsers"] =
       new Ext.BasicDialog("dialogAdministerUsers", {
         modal : false,
         width : 700,
         height : 310,
         shadow : true,
         shadowOffset : 10,
         proxyDrag: true
       }
  );
}
```

A quick check is done to ensure the dialog box hasn't already been created, and then a new `BasicDialog` instance is created. The first argument to the function is the ID of the `<div>` in `index.jsp` that contains the markup for the dialog box. The rest of the arguments define the look and feel of the dialog box, including its width and height, whether it has a shadow, how big that shadow is, whether it's modal or modeless (modeless means it will block all other UI functions until dismissed), and whether when dragged around it is the full dialog box or just a ghost version (when true). We keep a reference to the dialog box in an array in the timekeeper instance so we can manipulate it later, and that's all there is to it.

There are three other dialog boxes created: the Administer Projects, Manage Projects, and Using Timekeeper dialog boxes. But, as their code is virtually identical, we won't go over them here.

init() Now we come to that `init()` method I mentioned. This one is called `onLoad()`, which means it will be called **after** `initUI()` is called, and **after** all the contents for the page have loaded.

Before that though, let's take another look at the app as it might appear at this point, shown in Figure 9-7.

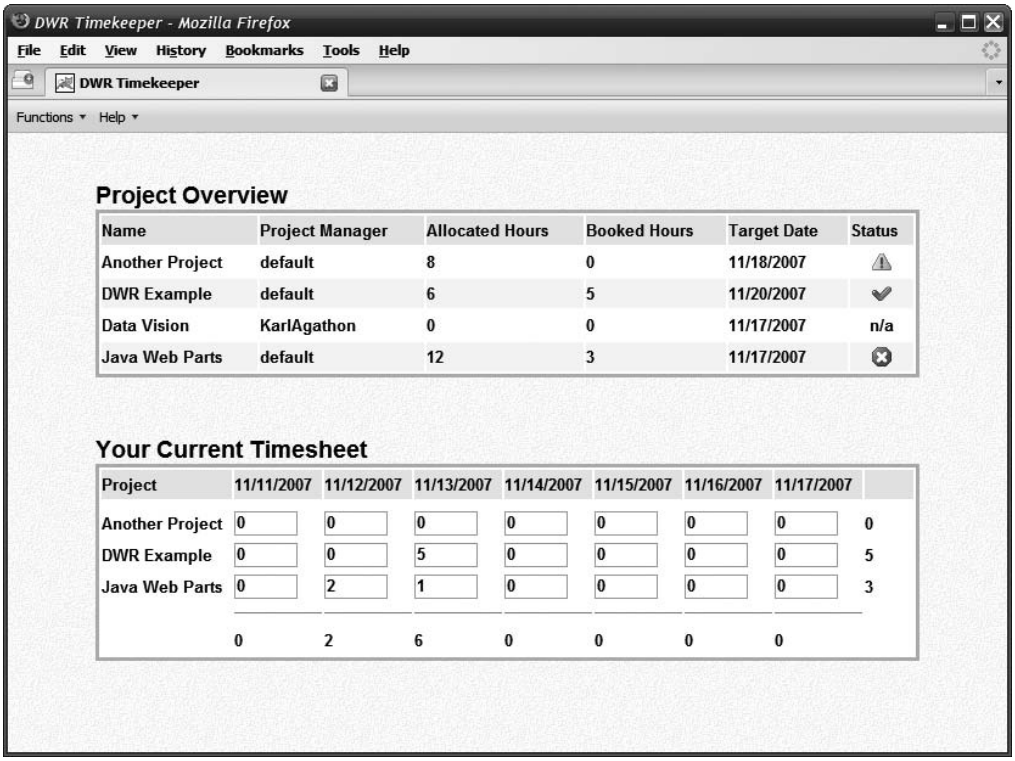

DWR Timekeeper - Mozilla Firefox
File Edit View History Bookmarks Tools Help
DWR Timekeeper
Functions Help
Project Overview
Name Project Manager Allocated Hours Booked Hours Target Date Status
Another Project default 8 0 11/18/2007
DWR Example default 6 5 11/20/2007
Data Vision KarlAgathon 0 0 11/17/2007 n/a
Java Web Parts default 12 3 11/17/2007
Your Current Timesheet
Project 11/11/2007 11/12/2007 11/13/2007 11/14/2007 11/15/2007 11/16/2007 11/17/2007
Another Project 0 0 0 0 0 0 0 0
DWR Example 0 0 5 0 0 0 0 5
Java Web Parts 0 2 1 0 0 0 0 3
0 2 6 0 0 0 0

Figure 9-7. *Another look at the home page, this time with project status icons in view*

In this figure, as opposed to Figure 9-2, which depicts the home page, you can see the icons showing the status of the projects here. We'll see how this status is determined later.

Anyway, moving on, the `init()` method begins like so:

```
this.init = function(inUserID) {
dwr.engine.setActiveReverseAjax(true);
```

Nothing special, we've seen reverse Ajax turned on before. Note the `inUserID` argument though. Recall in `index.jsp` we saw that the call to `init()` includes the user ID retrieved from session being passed in. I said then that we need to store that information client side, so this is the other half of the equation.

The first real work is to retrieve from the server a list of all users known to the application, and the `UserDAO` class provides that functionality:

```
UserDAO.listUsers(
  { async : false,
       callback : function(inResp) {
         timekeeper.users = inResp;
         timekeeper.currentUser = timekeeper.getUserByID(inUserID);
       }
  }
);
```

The array of `User` objects returned is stored in `timekeeper.users`, and we also at this point call the `getUserByID()` helper function so that we can get and store the `User` object corresponding to the logged in user.

Following that is some conditional logic dealing with enabling menu items:

```
if (timekeeper.currentUser.isAdministrator) {
  timekeeper.functionsMenu.items.get("functionsAdministerUsers").enable();
  timekeeper.functionsMenu.items.get(
      "functionsAdministerProjects").enable();
}
if (timekeeper.currentUser.isProjectManager) {
  timekeeper.functionsMenu.items.get("functionsManageProjects").enable();
}
```

Recall in `initUI()` that the menu items were by default disabled. Now, based on whether the user is an administrator or a project manager, we enable the associated menu items and make those capabilities available to that user.

The next step is a call to the `updateData()` method of the `timekeeper` object. What this method does will be seen in a bit, but to give you a preview, it updates data! (Yes, you now have implicit permission to kick me in the shin if you ever meet me in public, because really, what else could you do to someone that actually tries to make that joke work?!?)

The final bit of work to do in this method has to do with the timesheet shown on the home page, namely, setting the dates on the headers appropriately:

```
var today = new Date();
for (var i = 0; i < 7; i++) {
  var d = new Date();
  d.setTime(today.getTime() - (timekeeper.oneDay * i));
  d = (d.getMonth() + 1) + "/" + d.getDate() + "/" + d.getFullYear();
  dwr.util.setValue("divHome_bookTimeList_today_" + i, d + "  ",
      { escapeHtml : false }
  );
}
```

The way it works is this: we assume that time travel doesn't actually exist, at least, travel to the **future**, and therefore, booking time in advance of a day that hasn't come to pass yet **probably** doesn't make too much sense (yes, there are situations, like pending vacations,

where this makes sense, so consider this an opportunity for an enhancement suggestion!). So, with Einstein kicked out of the equation, we want the last date shown on the timesheet, the one all the way on the right, to be today's date, and then we want the six previous days to the left of it so that time can be booked for a week at a time (we're of course going with the American definition of "work week" here where there simply are no off days). So, we begin by creating a `Date` object, then setting it to today's date (using the `getTime()` method of the `Date` class) **minus** the number of milliseconds in a single day (which is the value of the `oneDay` field, which has a value of 1000 * 60 * 60 * 24) times the value of the loop value. The first loop iteration, where `i=0`, will result in the `Date` object having today's date (even **I** know this bit of math: anything times zero is zero), and so on down the line, filling in the headers.

menuClickHandler() Any time a menu item is clicked, the event results in this method being called. The argument passed to this method is the item itself, so we can retrieve the `text` attribute and `switch` on it to determine which item was clicked and react accordingly. Since all the `case` statements are essentially the same, save one, I'll just show the first one and you can be assured you've pretty much effectively seen the rest:

```
case "Administer Users": {
  timekeeper.dialogs["dialogAdministerUsers"].show(
    dwr.util.byId("divSource"));
break; }
```

So, having identified the menu item that was clicked, we get the handle to the dialog box that was created in `initUI()` from the dialog box array and call its `show()` method, passing it a reference to an element on the page with the ID `divSource`. The purpose of that is that when the dialog box is shown, it will perform an animated open effect, and to do so it requires a source element on the page to know where to start the animation from. You'll recall from `index.jsp` that `divSource` is a hidden `<div>` in the upper-left corner of the page, so mission accomplished, the animation can proceed from that point (the fact that it's hidden has no bearing, it still has X and Y coordinates on the page, which is what the animation effect needs).

The next `case` is slightly different, and that is the one for showing the About Timekeeper dialog box:

```
case "About Timekeeper": {
  Ext.MessageBox.alert("About Timekeeper",
    "DWR Timekeeper v1.0" +
    "<br><br>" +
    "By Frank W. Zammetti" +
    "<br><br>" +
    "Practical DWR 2 Projects" +
    "<br><br>" +
    "ISBN: 1-59059-941-1" +
    "<br><br>" +
    "Published In January 2008" +
    "<br><br>" +
    "<a href=\"http://apress.com/book/search?" +
    "searchterm=zammetti&act=search\">" +
    "Click here for details" +
```

```
    "</a><br>", function(){}
  );
break; }
```

Recall that in `index.jsp`, there was no markup for the About Timekeeper dialog box. That's because it's built programmatically here, if for no other reason than to demonstrate doing that. Here too we are using a different type of dialog box, namely a `MessageBox.alert`-type dialog box. This is akin to the standard JavaScript `alert()` pop-up, just a little prettier, and more importantly, it uses a lightbox effect like we've seen previously. This can be seen in Figure 9-8.

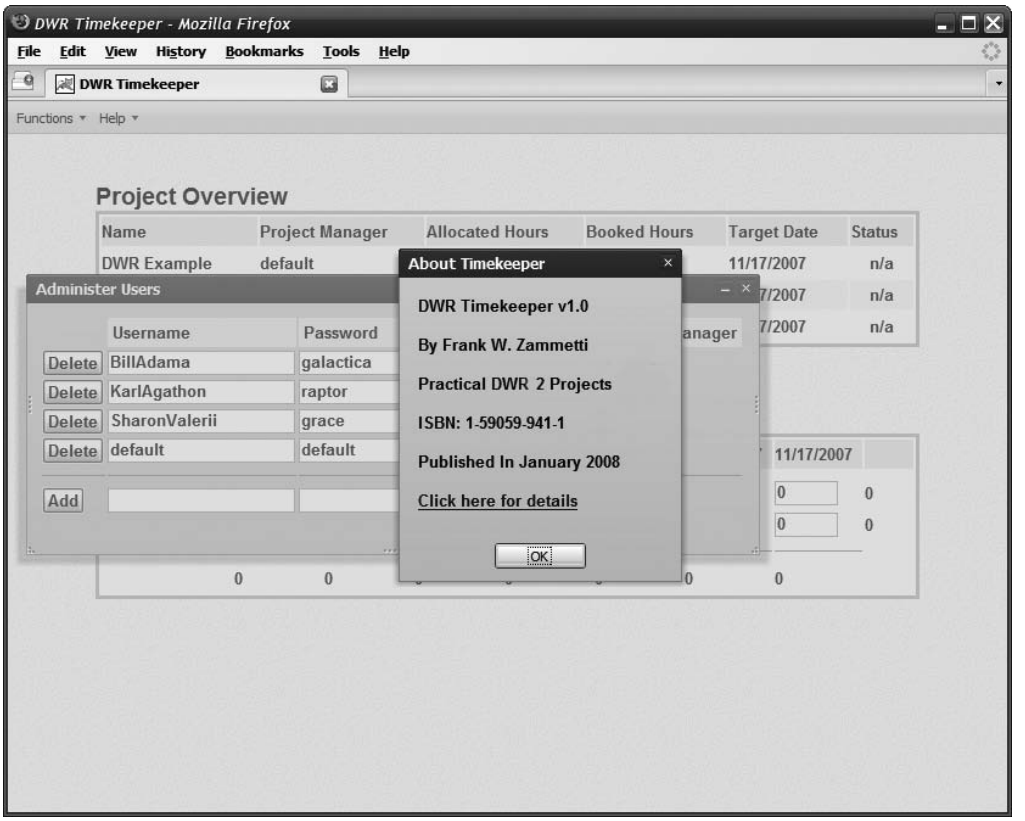


Figure 9-8. *The About pop-up*

Note how everything except the About pop-up itself is dimmed out, and how it has an OK button to dismiss it. Nothing else on the UI can be touched until this pop-up is dismissed, just like any lightbox.

updateData() Remember that `updateData()` method I mentioned earlier, and made an über-lame joke about? Well, here it is now:

```
this.updateData = function() {

  dwr.engine.beginBatch();
  timekeeper.updateUserList();
  timekeeper.getTimesheetItems();
  timekeeper.updateProjectList();
  dwr.engine.endBatch();

} // End updateData().
```

OK, so here's something new and exciting in DWR land! As you can see, there are three calls to methods of the `timekeeper` object being done. As it turns out, each of those makes a DWR call to the server. Normally, they would all fire individually and asynchronously, meaning three separate requests to the server, three separate threads of execution on the browser (as far as the network requests go, JavaScript is still always single-threaded, so only one callback at a time would ever execute), and so on. But, DWR offers one of the coolest features going: call batching. What this does is it rolls all the DWR requests between the call to `dwr.engine.beginBatch()` and `dwr.engine.endBatch()`, whether those remote calls are literally between those two statements or contained in functions called between those statements as is the case here, and combines them all into a single call. A **single** request is made to the server, and the DWR servlet deals with making the individual calls to the server-side objects. It then **combines** the responses and returns it. The DWR client-side code then calls the callbacks specified in the remote calls, in the sequence they appear in the batch. That is incredibly cool because it allows you to create much more efficient applications network-wise and cuts down resource utilization all around. Imagine trying to do this yourself with basic Ajax code, and you'll see how great a capability it is. Ajax calls tend to be one-off things, which is why we're only just now seeing this capability in action in this application: it (arguably at least) wasn't needed anywhere else. Here though, it makes for a more streamlined and efficient call mechanism.

Naturally, the details of those three methods will be gone over soon, but I wanted to explain the batching in isolation from that so it was clear and you really recognize just how useful that capability can be. It really is one of the coolest features DWR has to offer in my opinion.

showMessage() As you play with the application, you will have noticed that when updates are made to the server, there is a nice little message that slides down from the top, an example of which (minus the sliding naturally, which would be a neat trick on the printed page!) can be seen in Figure 9-9.

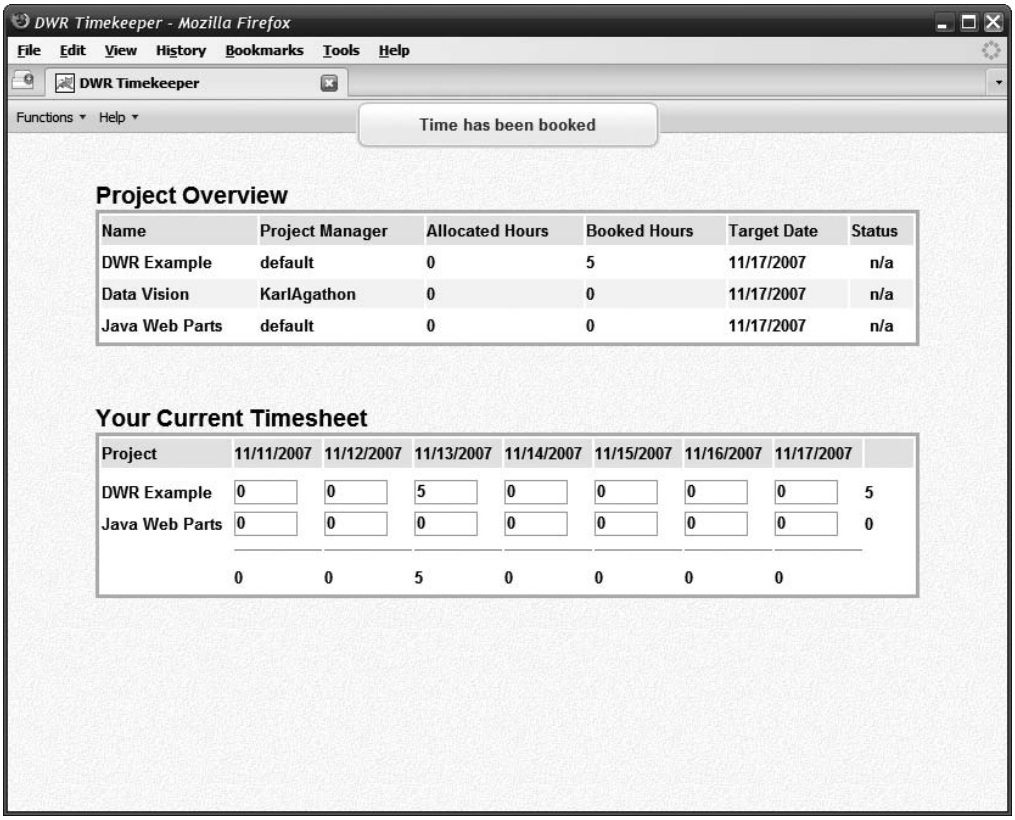

DWR Timekeeper - Mozilla Firefox
File Edit View History Bookmarks Tools Help
DWR Timekeeper
Functions Help
Time has been booked
Project Overview
Name Project Manager Allocated Hours Booked Hours Target Date Status
DWR Example default 0 5 11/17/2007 n/a
Data Vision KarlAgathon 0 0 11/17/2007 n/a
Java Web Parts default 0 0 11/17/2007 n/a
Your Current Timesheet
Project 11/11/2007 11/12/2007 11/13/2007 11/14/2007 11/15/2007 11/16/2007 11/17/2007
DWR Example 0 0 5 0 0 0 0 5
Java Web Parts 0 0 0 0 0 0 0 0
0 0 5 0 0 0 0

Figure 9-9. *The slide-down message can be seen at the top.*

The code that accomplishes that is as follows:

```
this.showMessage = function(inText) {

  if (!timekeeper.msgCt) {
    timekeeper.msgCt = Ext.DomHelper.insertFirst(document.body,
      {id : "msg-div"}, true);
  }
  timekeeper.msgCt.alignTo(document, "t-t");
  var m = Ext.DomHelper.append(timekeeper.msgCt,
    { html :
      "<div class=\"msg\"><div class=\"x-box-tl\">" +
      "<div class=\"x-box-tr\"><div class=\"x-box-tc\"></div></div></div>" +
      "<div class=\"x-box-ml\"><div class=\"x-box-mr\">" +
      "<div class=\"x-box-mc\"><center>" + inText +
      "</center></div></div></div>" +
      "<div class=\"x-box-bl\"><div class=\"x-box-br\">" +
      "<div class=\"x-box-bc\"></div></div></div></div>"
```

```
    }, true);
  m.slideIn("t").pause(2).slideOut("t", {remove : true});

} // End showMessage().
```

This code is, with some minor modifications, taken from one of the Ext JS example applications. In a nutshell, it's really just creating some markup, inserting it into the DOM using Ext JS's `DomHelper.append()` method, and then using a chain of effects to animate it. This chain includes a call to `slideIn("t")`, where "t" means top; next is a two-second pause; and finally, a `slideOut()` call occurs, sliding the message up to the top, and the remove flag is set to `true` to indicate we want the element created to be removed from the DOM afterwards. There is also, before all that actually, a check of the `msgCt` field to see whether a message already exists, and if so an element is inserted before the existing one. The net effect of this is that multiple messages can actually be displayed at once; they will simply "stack up" on the top of the screen and slide out of view one by one, to the beat of their own drum so to speak. To see this in action, go update some hours on your timesheet in rapid succession. You should be able to get two to three messages at the same time on screen. It's actually pretty fun to watch in action!

Methods Pertaining to Users

Now we come to a batch of methods pertaining to users, and mostly that means administration of users. First, it might be helpful to take a quick look at the Administer Users dialog box, since that's where most of these methods are used. That dialog box is shown here in Figure 9-10.

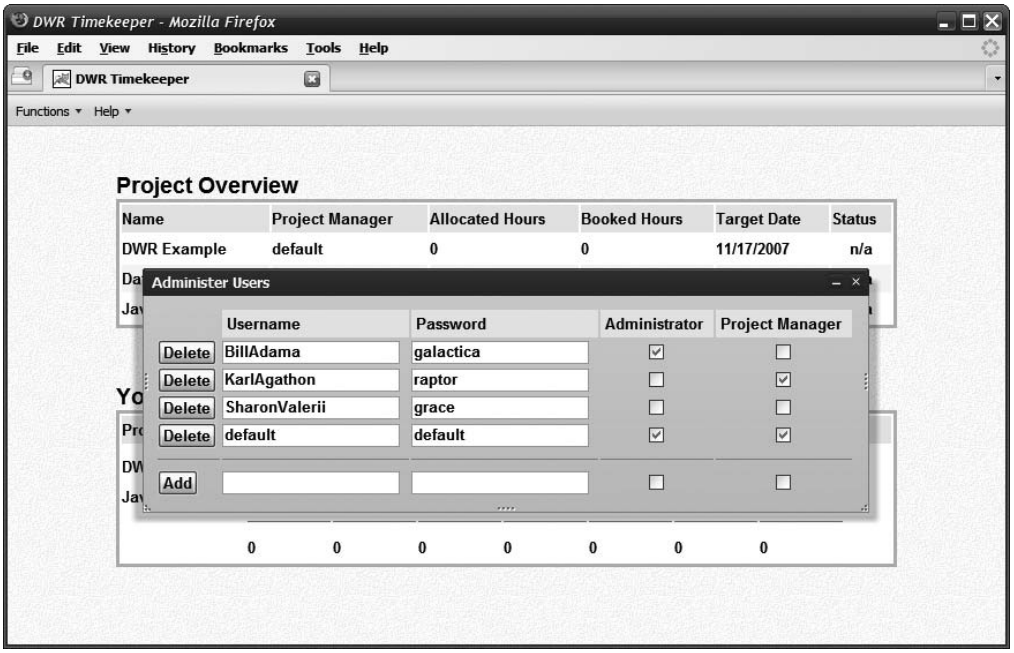


Figure 9-10. *The Administer Users dialog box*

The basic structure of this dialog box is that there is a list of the existing users and their attributes, and below the dividing line is where a new user can be created. As is true of all the dialog boxes in the application, as well as the timesheet on the home page, any edits to any of the fields are recorded real time on the server (where real time means when focus leaves the field). Also, with the help of reverse Ajax, any changes made by **other** users will be, more or less, immediately replicated out to all other users of Timekeeper that have it open at the time.

But, first things first . . . remember that `updateData()` method and how it called three other methods, one of which was `updateUserList()`? Well, that is in fact our first stop!

updateUserList() OK, so now we come to our first actual bit of code dealing with users, and I'd like to begin by pointing out that you'll find a number of spots in the following code that contain `...SNIP...` (you'll see this in later code as well). When you see this, it indicates that I've removed a chunk of code that is either a duplicate of the line that precedes it, or else continues the pattern begun in the previous two lines. This is yet another space-saving technique employed to try and cut the length of the chapter down a little.

Anyway, the method we're discussing now is responsible for getting an updated list of users, and then re-creating the user lists on the Administer Users dialog box. The code of that method is this:

```
this.updateUserList = function() {

  UserDAO.listUsers(
    {
      callback : function(inResp) {
        timekeeper.users = inResp;
        dwr.util.removeAllRows("divAdminUsers_userList");
        var cellFuncs = [
          function(data) { return data.split("~~")[0]; },
          function(data) { return data.split("~~")[1]; },
          ...SNIP...
        ];
        for (var i = 0; i < inResp.length; i++) {
          dwr.util.addRows("divAdminUsers_userList",
            [
              "<input type=\"button\" value=\"Delete\" " +
                "onClick=\"timekeeper.deleteUser('" +
                inResp[i].id + "', '" + inResp[i].username + "');\">~~" +
              "<input type=\"text\" " +
                "value=\"" + inResp[i].username + "\" " +
                "onBlur=\"timekeeper.updateUser('username', '" +
                inResp[i].id + "', this.value);\"" +
                ">~~" +
              "<input type=\"text\" " +
                "value=\"" + inResp[i].password + "\" " +
                "onBlur=\"timekeeper.updateUser('password', '" +
                inResp[i].id + "', this.value);\"" +
                ">~~" +
              "<center><input type=\"checkbox\" " +
```

```
            (inResp[i].isAdministrator ? "checked" : "") + " " +
            "onBlur=\"timekeeper.updateUser('isAdministrator', '" +
            inResp[i].id + "', this.checked);\"" +
            "></center>~~" +
          "<center><input type=\"checkbox\" " +
            (inResp[i].isProjectManager ? "checked" : "") + " " +
            "onBlur=\"timekeeper.updateUser('isProjectManager', '" +
            inResp[i].id + "', this.checked);\"" +
            "></center>"
        ],
        cellFuncs, { escapeHtml : false }
      );
    }
    dwr.util.addRows("divAdminUsers_userList",
      [
        "<hr size=\"1\" color\"#000000\">~~" +
        ...SNIP...
      ],
      cellFuncs, { escapeHtml : false }
    );
    dwr.util.addRows("divAdminUsers_userList",
      [
        "<input type=\"button\" value=\"Add\" " +
          "onClick=\"timekeeper.addUser();\">" + "~~" +
        "<input type=\"text\" " +
          "id=\"divAdminUsers_add_username\">" + "~~" +
        "<input type=\"text\" " +
          "id=\"divAdminUsers_add_password\">" + "~~" +
        "<center><input type=\"checkbox\" " +
          "id=\"divAdminUsers_add_isAdministrator\"></center>~~" +
        "<center><input type=\"checkbox\" " +
          "id=\"divAdminUsers_add_isProjectManager\"></center>"
      ],
      cellFuncs, { escapeHtml : false }
    );
  } // End callback function.
 } // End DWR options object.
);

} // End updateUserList().
```

The first thing we see is the call to the server-side `UserDAO.listUsers()`. This retrieves a list of `User` objects. With that list in hand, we remove all rows in the `<tbody>` of the table on the Administer Users dialog box. Next, we create the `cellFuncs` array as we've seen previously. Then, we iterate over the list of users and add a row to the table for each.

The complexity here is that there are form elements in each row allowing us to edit a given user. The code itself isn't really all that complex; it amounts to nothing but a giant string construction to create the markup for the row, although it looks like there's more to it than

that. Remember that the call to `dwr.util.addRows()` expects a string, and the `cellFuncs` will be splitting that string up on the ~~ character sequence, so every time you see ~~, you are seeing a table cell boundary in effect.

Going over each and every form element wouldn't be all that beneficial, but I suggest you do it at least this initial time to get a feel for it because you're going to see this same basic structure repeated a few more times throughout this code. In general, each element calls some method of the `timekeeper` instance in response to the `onBlur` event, and most of them pass as arguments to that method either the value of the form field itself or some piece of information from the `User` object (which is `resp[i]`, since we're iterating over an array of `User` objects), and sometimes both. There is also the Delete button, which again simply calls the appropriate method of `timekeeper`.

After that loop is another `dwr.util.addRows()` call, this time essentially creating a row where each cell contains an `<hr>` element, and that's how the dividing line gets into the table. Following that is the addition of the row for adding a new user, which is conceptually the same as any of the rows for editing existing users, except that nothing is done `onBlur` this time; it's all triggered off clicking the Add button.

addUser() The method called to add a user is fairly simple, and consists of the following code:

```
this.addUser = function() {

  var username = dwr.util.getValue("divAdminUsers_add_username");
  var password = dwr.util.getValue("divAdminUsers_add_password");

  if (username == "") {
    alert("Please enter a username");
    return;
  }
  if (password == "") {
    alert("Please enter a password");
    return;
  }

  UserDAO.addUser(
    username, password,
    dwr.util.getValue("divAdminUsers_add_isAdministrator"),
    dwr.util.getValue("divAdminUsers_add_isProjectManager"),
    {
      callback : function(inResp) {
        timekeeper.showMessage("User has been added");
        timekeeper.updateData();
      }
    }
  );
} // End addUser().
```

Once we retrieve the username and password entered using `dwr.util.getValue()`, we do some quick checks to make sure both were entered, and abort if not. Then it's a simple call to

the `addUser()` method of the `UserDAO` server object. Upon return of this call we show a slide-down message indicating the user was added, and we call `updateData()`. We need to do this because every list of users anywhere in the application needs to be updated now potentially, and this is an admittedly not very efficient way to achieve that, but it makes for very simple code, so the trade-off is worth it I think.

updateUser() Updating an existing user is only marginally more complex than adding a new one, as you can see from the following code:

```
this.updateUser = function(inFieldToUpdate, inID, inNewValue) {

  if (inFieldToUpdate == "isProjectManager") {
    for (var i = 0; i < timekeeper.projects.length; i++) {
      if (inID == timekeeper.projects[i].projectManager) {
        timekeeper.updateData();
        alert("User is assigned as project manager to project \"" +
          timekeeper.projects[i].name + "\"\n\nPlease unassigned the " +
          "user first.");
        return;
      }
    }
  }

  UserDAO.updateUser(inFieldToUpdate, inID, inNewValue,
    {
      callback : function(inResp) {
        timekeeper.showMessage("User has been updated");
        timekeeper.updateData();
      }
    }
  );

} // End updateUser().
```

The first argument to this method is what field is being updated. Doing it this way allows us to not have to have more than one update method. The first thing that's done is if the project manager check box is being changed, in the case of making the user no longer a project manager, we need to be sure that user isn't assigned as a project manager on any project. So, we iterate over the list of projects maintained in the `timekeeper.projects` field, and for each we see whether the user being changed is project manager for that project. If so, `updateData()` is called, which effectively resets any edits the user made, including the change to the project manager check box, and we pop up an alert saying it couldn't be done.

Assuming the update can proceed, it's again a simple call to another method of the `UserDAO` server object, `updateUser()` this time around. We pass it what field is being updated, the ID of the user, and the new value. Upon return we show a message indicating success, call `updateData()` once more to get the new value out there, and make any appropriate changes to any user lists that may need to be changed.

deleteUser() Deleting a user is a relatively easy operation, similar in at least one regard to updating a user. The code for the pertinent method is here:

```
this.deleteUser = function(inID, inUsername) {

  if (inID == timekeeper.currentUser.id) {
    alert("You cannot delete yourself");
    return;
  }
  for (var i = 0; i < timekeeper.projects.length; i++) {
    if (inID == timekeeper.projects[i].projectManager) {
      alert("User is assigned as project manager to project \"" +
        timekeeper.projects[i].name + "\"\n\nPlease unassigned the " +
        "user first.");
      return;
    }
  }

  if (confirm(
    "Are you sure you want to delete the user " + inUsername + "?")) {
    UserDAO.deleteUser(inID,
      {
        callback : function(inResp) {
          timekeeper.showMessage("User has been deleted");
          timekeeper.updateData();
        }
      }
    );
  }

} // End deleteUser().
```

It starts with one simple check: never allow a user to delete his or her own ID as suicide booths[3] aren't my bag. We then perform the same check as when updating a user to ensure we don't delete a user who is assigned as project manager to any projects (most projects can die on their own just fine without losing their supposed leader without a replacement). Finally, after a standard "are you sure you aren't doing something really stupid" confirmation, we call `deleteUser()` on the `UserDAO` object on the server to be sure to do the same message and data update as always, and the user is officially gone from the system at that point.

getUserByID() The `getUserByID()` method is extremely simple, so I think simply describing it will suffice: it iterates over the `timekeeper.users` array of `User` objects and finds the one with

3. Suicide booths are something that have been seen in the great show *Futurama* a number of times. Simply put: you step in, you put your quarter in the slot, you select your method of demise, and the booth dispatches you with great enthusiasm. Even gets rid of the leftovers, so no mess, and no fuss! Eh, call me crazy, but institutionalized suicide at the touch of a button just doesn't strike me as quite the best idea (but it's funny as can be on the show!).

an ID matching that of the argument passed in, if there is such a `User` object, and returns it, or `null` if not found. That's it, very quick and simple. But, it's a function needed elsewhere, so it had to be mentioned, regardless of its simplicity (and obviousness really).

isUserAssignedToProject() Just like `getUserByID()`, `isUserAssignedToProject()` is a very simple method: given a `Project` object, see whether a given user ID is contained in the list of assigned users and return `true` if so, `false` if not. The list of assigned users, which is the `usersAssigned` field of a `Project` object, is a comma-separated list of user IDs, so the string is first split, the resultant array is iterated over, and a simple == check is performed. This is another very simple but necessary method.

Methods Pertaining to Projects

Now we'll look at the methods that deal with projects, but even before we do that, let's take a quick look at the Administer Projects dialog box, as shown in Figure 9-11.

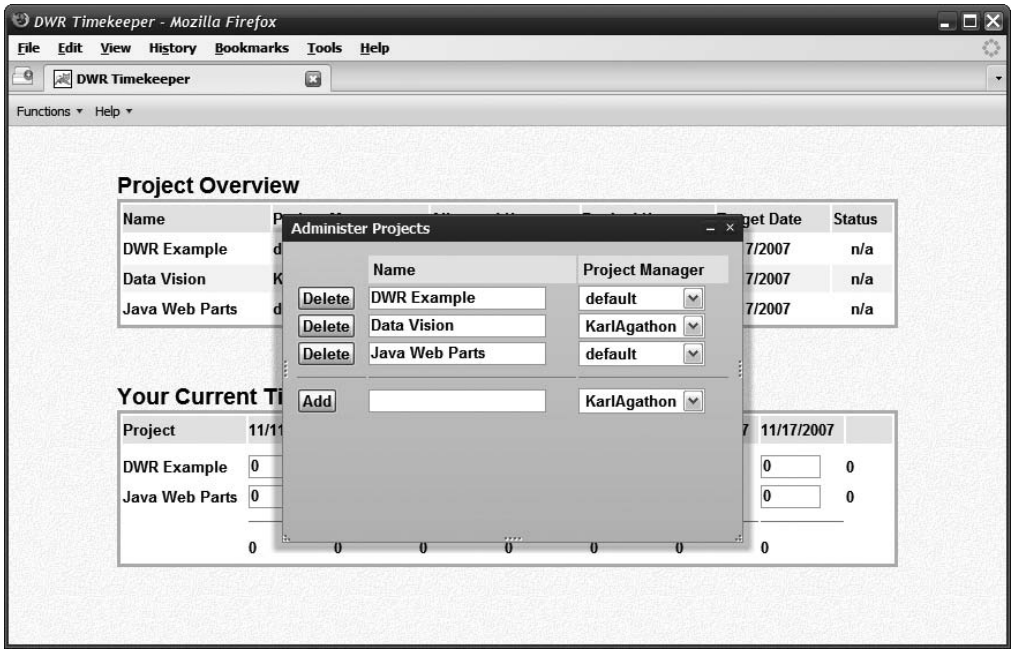


Figure 9-11. *The Administer Projects dialog box*

As you can see, it's very much similar to the Administer Users dialog box we already looked at, and in fact that's going to wind up being a common refrain in this section: the code for dealing with projects and users is very similar overall.

But, before that's true, let's deal with the case where it's not, and also deal with the single largest chunk of code we'll look at in this project, and that's the `updateProjectList()` method. (I highly suggest grabbing a cold drink and getting in a comfortable sitting position for this next section, as it's rather lengthy!)

updateProjectList() The reason there's so much code in this method is that there's quite a bit of work to be done (wow, that really does sound like something right out of a George W. Bush press conference, doesn't it?!). There are quite a few lists of projects throughout the application, and all need to be updated at the same time when updated data is retrieved from the server. Let's start from the beginning though:

```
ProjectDAO.listAllProjects(
  {
    callback : function(inResp) {
      dwr.util.removeAllRows("divHome_projectOverviewList");
      dwr.util.removeAllRows("divHome_bookTimeList");
      dwr.util.removeAllRows("divAdminProjects_projectList");
      dwr.util.removeAllRows("divManageProjects_projectList");
      timekeeper.projects = inResp;
      var dayTotals = new Array();
      for (var j = 0; j < 7; j++) {
        dayTotals.push(0);
      }
```

This is how the method begins, with a call to the `listAllProjects()` method of the `ProjectDAO` object on the server. The callback, which is where all the code that follows is encapsulated, begins by clearing the four tables that are the list of projects. There's the summary/overview table on the home page, the timesheet for the user, and of course the Administer Projects and Manage Projects dialog boxes. The next thing that's done is to create an array of all zeros representing the days on the timesheet.

Next up is this chunk:

```
      var altRow = false;
      for (var i = 0; i < inResp.length; i++) {
        var project = inResp[i];
        var targetDate = (project.targetDate.getMonth() + 1) + "/" +
          project.targetDate.getDate() + "/" +
          project.targetDate.getFullYear();
        var isUserAssignedToProject = timekeeper.isUserAssignedToProject(
          project, timekeeper.currentUser.id);
```

This is where we begin to iterate over the list of projects returned by the server. The first task is to get the target date of the project in a displayable MM/DD/YYYY format. Next, we need to determine whether the user is assigned to the project or not, and we know from our previous experience that there's an `isUserAssignedToProject()` method in the `timekeeper` object that will tell us this, so we're golden here.

Creating the Project Overview Summary Table on the Home Page

Next up is the first table creation, which is for the project overview summary on the home page:

```
        var cellFuncs = [
          function(data) { return data.split("~~")[0]; },
          function(data) { return data.split("~~")[1]; },
```

```
    ...SNIP...
  ];
  if (timekeeper.currentUser.isAdministrator ||
    timekeeper.currentUser.id == project.projectManager ||
    isUserAssignedToProject) {
    dwr.util.addRows("divHome_projectOverviewList",
      [
        project.name + "  ~~" +
        timekeeper.getUserByID(
          project.projectManager).username + "~~" +
        project.allocatedHours + "~~" +
        project.bookedHours + "~~" +
        targetDate + "~~" +
          timekeeper.calculateProjectStatus(project.allocatedHours,
          project.bookedHours, project.targetDate) + ""
      ],
      cellFuncs,
      { escapeHtml : false, altRow : altRow,
        rowCreator : function(inOptions) {
          var cn = null;
          if (inOptions.altRow) {
            cn = "cssTableAltRow";
          } else {
            cn = "cssTableRow";
          }
          var tr = document.createElement("tr");
          tr.setAttribute("class", cn);
          tr.setAttribute("valign", "middle");
          return tr;
        }
      }
    );
    altRow = !altRow;
  }
```

As you can see, we have a `cellFuncs` array, which you should be pretty familiar with by now. Next, we check to see whether the current user is an administrator, is the project manager for the current project being worked with, or is assigned to the project. If any of these conditions is true, the project will be displayed in the summary, so only projects that pertain to the user will be displayed. Once that's determined, it's just a `dwr.util.addRows()` call that you've seen a bunch of already. Note that this is again a string delimited with the ~~ sequence, which the `cellFuncs` functions split the string on to return the correct portion of the string for each cell in the table. Also note that the last cell in the table is where the project status icon image appears, and here there's a call to `calculateProjectStatus()`, which we'll see in a bit. In short, it returns a snippet of HTML that shows the appropriate image.

Something a little different here is the `rowCreator` element of the `options` object, which is the final argument passed to `dwr.util.addRows()`. This allows us to define how the `<tr>` element is created. The function we define here has the responsibility of instantiating a `<tr>`

element, manipulating it any way we want, and returning it. Here, we're using this feature to do row color striping, to ensure the rows in the table alternate colors.

Creating the Timesheet Entry Table on the Home Page

Next up is the creation of the timesheet table where the user can book time against projects:

```
var cellFuncs = [
  function(data) { return data.split("~~")[0]; },
  function(data) { return data.split("~~")[1]; },
  ...SNIP...
];
if (timekeeper.currentUser.id == project.projectManager ||
  isUserAssignedToProject) {
  var hoursBooked = new Array();
  var projectHours = 0;
  for (var j = 0; j < 7; j++) {
    var d = new Date();
    d.setTime(d - (timekeeper.oneDay * j));
    d.setHours(0, 0, 0, 0);
    var timesheetItem = timekeeper.getBookedTimeForProjectByDate(
      project.id, d)
    hoursBooked.push(timesheetItem);
    if (timesheetItem != null) {
      projectHours += timesheetItem.hours;
      dayTotals[j] += timesheetItem.hours;
    }
  }
  dwr.util.addRows("divHome_bookTimeList",
    [
      project.name + "  ~~" +
      "<input type=\"text\" size=\"3\" maxlength=\"2\" " +
        "value=\"" + (hoursBooked[6] ? hoursBooked[6].hours : "0") +
        "\" onBlur=\"timekeeper.saveTimesheetItem('" + project.id +
        "', 6, this.value);\">~~" +
      "<input type=\"text\" size=\"3\" maxlength=\"2\" " +
        "value=\"" + (hoursBooked[5] ? hoursBooked[5].hours : "0") +
        "\" onBlur=\"timekeeper.saveTimesheetItem('" + project.id +
        "', 5, this.value);\">~~" +
        ...SNIP...
      projectHours
    ],
    cellFuncs,
    { escapeHtml : false,
      rowCreator : function(inOptions) {
        var tr = document.createElement("tr");
        tr.setAttribute("class", "cssTableRow");
        tr.setAttribute("valign", "middle");
```

```
            return tr;
          }
        }
      );
    }
```

It starts with `cellFuncs`, same as always. Next is a check to see whether the current project is one the user is a project administrator for or is assigned to. Those are the only cases where a user should be able to book time to a project (an administrator, we assume, doesn't automatically work on projects; he or she would have to be assigned like any other user would). Once that's determined, it's on with the work.

That work begins with getting the number of hours booked to this project for this period and this user, and also calculating the total hours for this project in this period. For this to work, we do a loop that corresponds to the days in the table headers, but it does so in a seemingly strange way. Remember that the headers are laid out with the current date all the way to the right, and the dates descend when read from right to left starting with today's date. So, with each loop iteration, we begin with today's date, and from that we subtract the number of days of the loop iteration. Then, a call to the `getBookedTimeForProjectByDate()` method gives us the value that should appear in the timesheet. We also store this value in the `hoursBooked` array for later, and assuming there was a `TimesheetItem` for the date, we also accumulate the number of hours for the project and for all projects on the given date.

After that, it's a typical call to `dwr.util.addRows()`. Note that when the string is constructed and the value is being put into the text box, there is a possibility that there is no element in the array corresponding to the date, so that's where that little bit of trinary logic comes into play: we only try to retrieve the `hours` attribute if the element pulled from the array is not null; otherwise, we'd have the possibility of a null pointer here.

You may have cause to wonder, with all these `cellFuncs` arrays doing the same thing, why not just use one over and over again? As it turns out, if the number of elements in the array doesn't match the number of cells in the table, it just doesn't work. I didn't spend any time trying to figure out why, I frankly just took it as a restriction you have to deal with, but it does appear to be the case. Sure, I could have had one and then just spliced elements out of it, or added new ones, to make it the size I needed in each case, but at the point you're writing that code, is it truly any better? I didn't really think so, hence simply creating a new version each time as necessary.

Creating the Table of Projects for the Project Administration Dialog Box

Moving right along, we come to the next block of code, which builds the project list in the Administer Projects dialog box:

```
var cellFuncs = [
  function(data) { return data.split("~~")[0]; },
  function(data) { return data.split("~~")[1]; },
  function(data) { return data.split("~~")[2]; }
];
```

```
dwr.util.addRows("divAdminProjects_projectList",
  [
    "<input type=\"button\" value=\"Delete\" " +
      "onClick=\"timekeeper.deleteProject('" +
      project.id + "', '" + project.name + "');\">" + "~~" +
    "<input type=\"text\" size=\"20\" " +
      "value=\"" + project.name + "\" " +
      "onBlur=\"timekeeper.updateProject('name', '" +
      project.id + "', this.value);\">~~" +
    "<select id=\"divAdminProjects_edit_pm_" + project.id +
      "\" onBlur=\"timekeeper.updateProject('projectManager', '" +
      project.id + "', this.value);\"></select>"
  ],
  cellFuncs, { escapeHtml : false }
);
for (var j = 0; j < timekeeper.users.length; j++) {
  if (timekeeper.users[j].isProjectManager) {
    dwr.util.addOptions("divAdminProjects_edit_pm_" + project.id,
      [ { value : timekeeper.users[j].id,
          text : timekeeper.users[j].username } ],
      "value", "text"
    );
  }
}
dwr.util.setValue("divAdminProjects_edit_pm_" + project.id,
  project.projectManager);
```

The actual building of the table should pretty well be old hat to you by now, so we'll skip a lengthy explanation. Something more interesting abounds, however, with the line beginning the loop. For each project in the list, there needs to be a list of project managers for the administrator to select from. To do this is simply a matter of iterating over all the users in the `timekeeper.users` array, and seeing whether each is a project manager. For those who are, we use the `dwr.util.addOptions()` function to add it to the `<select>` element with the ID `divAdminProjects_edit_pm_XXXX`, where `XXXX` is the ID of the project.

Once the list of project managers is populated, the last task is to select the current project manager, if any, and that's where the final line with the `dwr.util.setValue()` comes in.

Creating the Table of Projects for the Project Management Dialog Box

Next up is a relatively lengthy bit of code that builds the project list on the Manage Projects dialog box. As it turns out, this is virtually identical to the chunk of code we just looked at.

```
if (project.projectManager == timekeeper.currentUser.id) {
  var cellFuncs = [
    function(data) { return data.split("~~")[0]; },
    ...SNIP...
  ];
  dwr.util.addRows("divManageProjects_projectList",
```

```
        [
          project.name + "~~" +
          "<input type=\"text\" size=\"3\" maxlength=\"3\" " +
            "value=\"" + project.allocatedHours + "\" " +
            "onBlur=\"timekeeper.updateProject('allocatedHours', '" +
            project.id + "', this.value);\">~~" +
          "<input type=\"text\" size=\"11\" maxlength=\"10\" " +
            "value=\"" + targetDate + "\" " +
            "onBlur=\"timekeeper.updateProject('targetDate', '" +
            project.id + "', this.value);\">~~" +
          "<select id=\"divManageProjects_edit_ua_" + project.id +
            "\" onBlur=\"timekeeper.updateProject('usersAssigned', '" +
            project.id + "', " +
            "dwr.util.getValue('divManageProjects_edit_ua_" +
            project.id + "').toString());\" multiple " +
            "size=\"4\"></select>"
        ],
        cellFuncs, { escapeHtml : false }
      );
      for (var j = 0; j < timekeeper.users.length; j++) {
        dwr.util.addOptions("divManageProjects_edit_ua_" + project.id,
          [ { value : timekeeper.users[j].id,
              text : timekeeper.users[j].username } ],
          "value", "text"
        );
      }
      var usersAssigned = project.usersAssigned;
      if (usersAssigned != "") {
        usersAssigned = usersAssigned.split(",");
        for (var j = 0; j < usersAssigned.length; j++) {
          dwr.util.setValue("divManageProjects_edit_ua_" + project.id,
            usersAssigned[j]);
        }
      }
    }
  } // End iteration over projects collection.
```

After the table is built, we have another similar loop as seen in the last chunk that adds all the users to a `<select>` element (not just project managers this time), and then as part of that loop we also select the user if he or she is assigned to the project. Overall, very much like the previous bit of code.

We're making good progress getting through this method, so let's keep that going. The previous blocks of code we've examined were concerned with building tables and the elements within those tables. At this point, however, the iteration over the collection of projects has ended, and the next bit of code is concerned with adding the lines onto those tables that allow the user to add new projects, specifically in the Administer Projects dialog box.

```
var cellFuncs = [
  function(data) { return data.split("~~")[0]; },
  function(data) { return data.split("~~")[1]; },
  function(data) { return data.split("~~")[2]; }
];
dwr.util.addRows("divAdminProjects_projectList",
  [
    "<hr size=\"1\" color\"#000000\">~~" +
    "<hr size=\"1\" color\"#000000\">~~" +
    "<hr size=\"1\" color\"#000000\">"
  ],
  cellFuncs, { escapeHtml : false }
);
dwr.util.addRows("divAdminProjects_projectList",
  [
    "<input type=\"button\" value=\"Add\" " +
      "onClick=\"timekeeper.addProject();\">" + "~~" +
    "<input type=\"text\" " +
      "id=\"divAdminProjects_add_name\">" + "~~" +
    "<select id=\"divAdminProjects_add_projectManager\"></select>"
  ],
  cellFuncs, { escapeHtml : false }
);
for (var j = 0; j < timekeeper.users.length; j++) {
  if (timekeeper.users[j].isProjectManager) {
    dwr.util.addOptions("divAdminProjects_add_projectManager",
      [ { value : timekeeper.users[j].id,
          text : timekeeper.users[j].username } ],
      "value", "text"
    );
  }
}
```

First, a row is added with <hr> elements in it to give us a dividing line between the existing projects and the add project row. Then, the add project row itself is added, in very much the same basic way that any other row is added.

Finally, once those two rows are added, only a single task remains, and that's to add the list of project managers to the <select> in the add project row, which is what the code inside the loop there does, just like we saw in the preceding code.

OK, only a single block of code remains to examine, concerned with added day totals to the timesheet table on the home page, and here it is:

```
var cellFuncs = [
  function(data) { return data.split("~~")[0]; },
  ...SNIP...
];
dwr.util.addRows("divHome_bookTimeList",
  [
```

```
      " ~~" +
      "<hr size=\"1\" color\"#000000\">~~" +
      ...SNIP...
      " "
    ],
    cellFuncs, { escapeHtml : false }
  );
  dwr.util.addRows("divHome_bookTimeList",
    [
      " ~~" + dayTotals[6] + "~~" + dayTotals[5] + "~~" +
      dayTotals[4] + "~~" + dayTotals[3] + "~~" + dayTotals[2] + "~~" +
      dayTotals[1] + "~~" + dayTotals[0] + "~~" + " "
    ],
    cellFuncs,
    { escapeHtml : false,
      rowCreator : function(inOptions) {
        var tr = document.createElement("tr");
        tr.setAttribute("class", "cssTableRow");
        tr.setAttribute("valign", "middle");
        return tr;
      }
    }
  );
```

First, a divider row is added, just like you've previously seen. Next, the row that has the totals for each day is added. Recall that earlier we saw that these values are stored in the `dayTotals` array, so it's a simple matter to construct this row now. We also use a custom `rowCreator` here so that we can change the alignment of the cells to center things, which I think looks a little better.

Whew, that was a ton of code to get through, but we made it! Thankfully, what remains isn't nearly as verbose or complex, although I don't think the code we just reviewed was really all that complex, but there was quite a bit of it.

One question you are almost certain to ask at this point is why I didn't break this method up into a couple of separate methods. To be very blunt, the answer is I probably should have! Whether breaking up a large method is the right thing to do is always a question you have to ask, I think. Sometimes it pretty clearly is, other times it pretty clearly isn't, and sometimes it's a borderline call. The only real reason I didn't break this one up is that this code is all self-contained and not really reusable, so having to mentally jump around from method to method is something I tend to prefer avoiding in such a case. It was also all basically a straight-through logic flow, not really anything in the way of branching to really confuse matters, so I didn't see it as simplifying anything. In the end though, if I had it to do all over again, I think I would have broken it up. That being said, I don't think it's a clear-cut answer either way.

---

addProject(), updateProject(), and deleteProject() These three methods are very much like their user counterparts, `addUser()`, `updateUser()`, and `deleteUser()`. Because of this, I've chosen to not show them here, or really go over them. Please have a quick look at them, but don't spend too much time on them, as you've effectively already seen them when looking at the user versions; just change the word "user" to "project" in all cases, and you've pretty much got it.

calculateProjectStatus() The last method related to projects is a bit different. It's the method that determines whether a project is late, on time, or borderline. Here's the code:

```
this.calculateProjectStatus = function (inAllocatedHours,
  inBookedHours, inTargetDate) {

    var today = new Date();
    var daysUntilTarget =
      Math.ceil((inTargetDate.getTime() - today.getTime()) /
      (timekeeper.oneDay));
    var hoursUntilTarget = daysUntilTarget * 8;
    var allocatedHoursLeft = inAllocatedHours - inBookedHours;
    if (inAllocatedHours == 0) {
      return "<center>n/a</center>";
    }
    if (allocatedHoursLeft < hoursUntilTarget) {
      return "<center>" +
        "<img src=\"img/ok.gif\">" + "</center>";
    }
    if (allocatedHoursLeft > hoursUntilTarget) {
      return "<center>" +
        "<img src=\"img/late.gif\">" + "</center>";
    }
    if (allocatedHoursLeft == hoursUntilTarget) {
      return "<center>" +
        "<img src=\"img/borderline.gif\">" + "</center>";
    }

} // End calculateProjectStatus().
```

The way it works is this: passed in is the number of hours allocated to a given project, the total number of hours booked to it so far for all assigned users, and its target completion date.

First, we calculate how many days there are until the target date. This is a simple matter of subtracting the number of milliseconds in the current date from the number of milliseconds in the target date and dividing by the number of milliseconds in a single day, conveniently found in the `oneDay` field of the `timekeeper` object, as we saw earlier. Next, we calculate how many hours there are until the target date. A simplifying assumption is made here: a work day is always 8 hours, no more and no less. With that assumption in hand, it's a simple multiplication of the days until the target date, just calculated, times 8.

Then, we take the number of hours allocated to the project and subtract the number of hours booked so far against the project, which yields how many hours that have been allocated remain in the project.

After all these calculations, we then have enough information in hand to actually determine the status of the project. First, a check for a simple case: if no hours have been allocated to the project yet, which is an acceptable condition, this calculation doesn't apply, so the string "n/a" is returned to indicate the project doesn't really have a status yet (at least none that this application cares about).

Next, if the number of remaining allocated hours is less than the number of hours left until the target date, we consider the project to be on time, or OK. In that case, we return a little snippet of markup that, when inserted into the table, will result in the `ok.gif` image being shown.

However, if the remaining number of allocated hours is greater than the number of hours remaining until the target date, the project is considered late and hence the `late.gif` image needs to be shown.

The final condition is if the number of remaining allocated hours equals the number of hours left until the target date, this project is considered borderline because any slip will result in it being late, and any time that gets us ahead of schedule makes it OK, so the `borderline.gif` image is applicable here.

Methods Pertaining to Timesheets

The next couple of methods deal with timesheets and timesheet items, which are entries in a user's timesheet for a given project on a given day.

getTimesheetItems() The first method we come across is `getTimesheetItems()`, and it's literally one line of code (albeit one a little more than what's generally considered a "single line"): a call to the `getTimesheetItems()` method of the `TimesheetItemDAO` server object. This method gets passed the ID of the current user, and the callback does nothing but store the return, which is an array of `TimesheetItem` objects, in the `timesheetItems` field of the `timekeeper` object. Really, that's it (so simple I didn't waste space showing it here)!

getBookedTimeForProjectByDate() The next method is `getBookedTimeForProjectByDate()`, and it's a method with a relatively long name, but thankfully not a ton of code:

```
this.getBookedTimeForProjectByDate = function(inProjectID, inDate) {

  inDate.setHours(0, 0, 0, 0);
  if (timekeeper.timesheetItems) {
    for (var i = 0; i < timekeeper.timesheetItems.length; i++) {
      if (timekeeper.timesheetItems[i].projectID == inProjectID &&
        timekeeper.timesheetItems[i].reportDate.getTime() ==
        inDate.getTime()) {
        return timekeeper.timesheetItems[i];
      }
    }
  }
  return null;

} // End getBookedTimeForProjectByDate().
```

In addition to the ID of the project to get booked time for, a date is passed in that is the date we want to look up. The first step is to essentially ensure that only the date portion will be important. When the server gets ahold of this, it will be doing database queries against date fields, and if we just used the current time, there is the risk that the queries won't find matches if the time portion of the database date field doesn't match. So, by setting the time to zero (with the poorly named `setHours()` method) and doing this when we save a `TimesheetItem` to the database as we'll see shortly, we know that all dates in the database have the same time component to the dates, and hence they can essentially be ignored since they'll always match.

Once that little bit of level setting is done, we see whether there are `TimesheetItem` objects stored for the current user. As it turns out, although the timesheet on the home page shows zeros in fields the user hasn't entered hours for, in the database there actually are no records for those dates until the user edits them. Therefore, during this method, the possibility exists that the array isn't populated yet, hence the check.

Once we are sure there are items in the array, we begin to iterate over it. Then, we look for the item that corresponds to the requested project, and where the date matches the date requested, and when it's found, we return it. Should it not be found, `null` is returned.

saveTimesheetItem() Saving a `TimesheetItem` is quite similar to saving a user or project, but not exactly. Let's have a look, shall we?

```
this.saveTimesheetItem = function(inProjectID, inDayDiff, inHours) {

  var d = new Date();
  d.setTime(d - (timekeeper.oneDay * inDayDiff));
  d.setHours(0, 0, 0, 0);
  var item = timekeeper.getBookedTimeForProjectByDate(inProjectID, d);
  if (item) {
    TimesheetDAO.updateItem(item.id, inHours,
      { callback : function(inResp) {
          timekeeper.showMessage("Time has been booked");
          timekeeper.updateData();
        }
      }
    );
  } else {
    TimesheetDAO.addItem(timekeeper.currentUser.id, inProjectID, d, inHours,
      { callback : function(inResp) {
          timekeeper.showMessage("Time has been booked");
          timekeeper.updateData();
        }
      }
    );
  }

} // End saveTimesheetItem().
```

So, to begin with, we create a `Date` object, and then a little bit of funkiness ensues. You see, the way the save works is that the caller of this method indicates what date to save not by specifying the date, but by specifying how many days prior to today's date is being saved. As weird as this seems, it actually allows the code to be **simpler** because of the way the timesheet table is laid out. Remember how the headers were populated, in reverse order when read left to right? That means you can essentially deal with them positionally, i.e., their offset from the rightmost date determines how many days prior to today's date it is. With that knowledge, it's a simple calculation to determine how to set the `Date` object: take the number of days back from today's date, multiplied by how many milliseconds are in a single day, subtract that value from the current date (in milliseconds), and set that result on the `Date` object, which will now have the date we're trying to save.

Next we see the time component leveling I mentioned in the previous method.

The next step is to determine whether there already exists a `TimesheetItem` for this project and date. If so, we're doing an update; if not, we're creating of a new item. This branching really only alters what method of the `TimesheetDAO` object gets called, as well as what arguments the method needs. Aside from that, it's basically the same flow.

TIME AND RELATIVE DIMENSIONS IN SPACE

TARDIS, an acronym for Time And Relative Dimensions In Space, is of course the phone booth–style space/time craft operated by the Doctor on the famous British sci-fi show *Dr. Who*. TARDIS is also what I should have called this project, but it only occurred to me on the last day of writing this chapter!

You may be wondering what relevance this has, and the answer is none! But, with all this date-related coding, I thought I'd point out a fabulous reference on the Web that I went to many times during the writing of this chapter: W3 Schools (`www.w3schools.com/default.asp`). This is a reference site for all sorts of things related to web development, including, but not limited to, JavaScript references, CSS references, HTML references, and Ajax references. It has a combination of reference material and "learning" articles, whatever your needs are.

I often find that doing a Google search for something like "date object reference" will bring me to this site, so now I simply keep a bookmark to it handy to save the typing! It's a stellar reference, and I highly suggest getting to know it, as it'll likely serve you very well.

The Server-Side Code

Having just completed the client-side code, it's now time to look at the server-side code that we've seen calls to throughout the client-side code. We'll begin with a little utility class, very uncreatively named `Utils`.

Utils.java

In Figure 9-12, you can see the rather paltry UML class diagram for the `Utils` class.

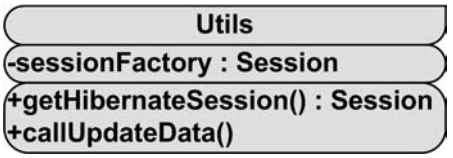


Figure 9-12. *UML class diagram of the* `Utils` *class*

The code for the class is as follows (minus imports, package statement, and comments):

```
public class Utils {

  private static SessionFactory sessionFactory;

  static {
    sessionFactory = new Configuration().configure().buildSessionFactory();
  } // End static initializer block.

  public static Session getHibernateSession() {
    return sessionFactory.getCurrentSession();
  } // End getHibernateSession().

  public static void callUpdateData() {
    WebContext wctx = WebContextFactory.get();
    String currentPage = wctx.getCurrentPage();
    ScriptBuffer script = new ScriptBuffer();
    script.appendScript("timekeeper.updateData();");
    Collection pages = wctx.getScriptSessionsByPage(currentPage);
    for (Iterator it = pages.iterator(); it.hasNext();) {
      ScriptSession session = (ScriptSession)it.next();
      session.addScript(script);
    }
  } // End callUpdateData().

} // End class.
```

First, a little bit of Hibernate knowledge for you to take in. Every operation you perform with Hibernate happens within the context of a `Session`. This allows you to let Hibernate deal with things like transactions, rollbacks, and all that good stuff. Any time you perform a function, you'll need to ask Hibernate for a session, and this is done via a `SessionFactory`.

Now, in most Hibernate applications, the `SessionFactory` is instantiated and configured exactly once, and that's what happens here by virtue of the static initializer block. The line

```
sessionFactory = new Configuration().configure().buildSessionFactory();
```

is what does it for us. This essentially uses smart defaults, which includes looking for our `hibernate.cfg.xml` file in the classpath and configuring the factory with the values in it.

So, back to actually performing an operation . . . It will always start with a call to the first actual method in the `Utils` class, `getHibernateSession()`, and as you can see, it's nothing but a call to the `SessionFactory` to get the current session and return that. The current session is tied to the executing thread, so it's perhaps not quite the best name for it, but if it helps, you can basically think of it as `getASession()` instead—that's about what it is conceptually from the standpoint of the code that uses this. With that session in hand, the calling code can now perform database operations, but that's getting ahead of things a bit. We'll see all that when we encounter our first DAO.

The last method in `Utils` is `callUpdateData()`, and it's a DWR-related method. In short, it's what performs our reverse Ajax. First, a `WebContext` instance is gotten, which we've seen previously. Next, a call to `getCurrentPage()` is made, which gives us the name of the current page being viewed by the caller. `ScriptBuffer` is then instantiated, and a method call to the `updateData()` method of the client-side `timekeeper` object is appended to it.

Next, a loop is begun that iterates over all the sessions currently "attached" to the page that we got the name of previously. Then, we call `addScript()` on each session.

So, what this does is simply find all the users viewing the current page (which will, of course, always be `index.jsp` since that's the only page in this app) and send a request via Comet to their browsers to call the `updateData()` method. Remember from our earlier look at the client code that this will trigger a batched call back to the server to update all the data across the application for that user. Pretty nifty, huh?

DESIGNING FOR EFFICIENCY (OR NOT!)

In this application, I've made some trade-offs in a couple of spots to make the code simpler at the cost of not being as efficient as possible. This is a trade-off you always have to make, most especially when dealing with Ajax, because there are quite a few factors that go into deciding what's "efficient" and what's not, what's "simple" and what's not.

This here is a good example: wouldn't it have been more efficient to send back just the data that had changed via Comet and then not have had to have the client make another call back to the server for updated data, and then rebuild everything? Yes, absolutely, without question that would have been more efficient! But, it also would have been exponentially more complex to implement. You'd have to figure out a data structure that makes sense to pass back, would have had to write the code to construct that structure, and then would have had to have a lot more code on the client side to deal with it. You have to make up your own mind, of course, but I decided the code being simpler and more easily comprehended and maintained was worth more than that bit of efficiency. I'm not saying it's the **right** answer, just that it's **my** answer.

User.java

The `User` class is a simple VO describing a user of the Timekeeper application. Just like `Project`, I won't be listing the source for this very simple class here, but will instead simply summarize the fields found in objects of this type in Table 9-2.

Table 9-2. *The Fields Contained in the* `User` *Class*

Data Type	Name	Description
Long	`id`	Unique ID of the user. This is automatically populated by Hibernate.
String	`username`	The user's username.
String	`password`	The user's password.
Boolean	`isAdministrator`	Flag: Is the user an administrator? If so, he or she can modify the parameters of a project such as target date, allocated hours, and assigned users.
Boolean	`isProjectManager`	Flag: Is the user a project manager? If so, he or she can create, edit, and delete projects and users.

UserDAO.java

The `UserDAO` class takes care of all database functions related to users and, naturally enough, `User` objects. The UML class diagram for this DAO can be seen in Figure 9-13.

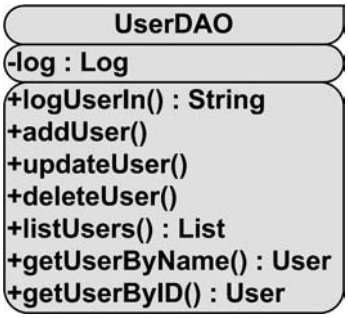


Figure 9-13. *UML class diagram of the* `UserDAO` *class*

logUserIn(): Can I Get a "Come on in, Friend"?

The first method we encounter in the `UserDAO` class is `logUserIn()`, which is called, not surprisingly, when the user attempts to log in.

```
@RemoteMethod
public String logUserIn(final String inUsername, final String inPassword,
  final HttpSession inSession) throws Exception {

  List users = listUsers();
  if (users.size() == 0) {
    addUser("default", "default", true, true);
  }
  User user = getUserByName(inUsername);
  if (user == null || !user.getPassword().equals(inPassword)) {
    return "Bad";
```

```
  } else {
    inSession.setAttribute("user", user);
    return "Ok";
  }

} // End logUserIn().
```

First, remember that Hibernate will be creating the database for us under the covers, which means that when this method is called for the very first time, there will be no users in the database, so there's never the possibility of logging in initially. To deal with this, I decided to have a single account created initially with a username and password of "default". So, the first thing that happens in this method is to get a list of all users by calling the `listUsers()` method, also of the `UserDAO`. If the size of the list returned is zero, a call to `addUser()` occurs to create that default user.

After this, whether that user was added or not, a call to `getUserByName()` is done, which does exactly what its name says, namely returning a `User` object given a specific username. If we don't get a `User` object back, or if the password passed in doesn't match that of the `User` object we got, the string `Bad` is returned to the client; otherwise `Ok` is returned. In the case of a successful logon, the `User` object is also added to session and is used in `index.jsp` as you saw previously.

addUser(): Because Life Is Better with Friends

In the `logUserIn()` method, we saw a call to `addUser()`, and now we'll get a chance to see what that method is all about:

```
@RemoteMethod
public void addUser(final String inUsername, final String inPassword,
  final boolean inIsAdministrator, final boolean inIsProjectManager)
  throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("addUser() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.debug("addUser() - inUsername = " + inUsername);
    log.debug("addUser() - inPassword = " + inPassword);
    log.debug("addUser() - inIsAdministrator = " + inIsAdministrator);
    log.debug("addUser() - inIsProjectManager = " + inIsProjectManager);
  }
  User user = new User();
  user.setUsername(inUsername);
  user.setPassword(inPassword);
  user.setIsAdministrator(inIsAdministrator);
  user.setIsProjectManager(inIsProjectManager);
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  session.save(user);
  session.getTransaction().commit();
```

```
  Utils.callUpdateData();
  if (log.isTraceEnabled()) {
    log.trace("addUser() - Exit");
  }

} // End addUser().
```

First, some logging is performed in case we need to do any debugging. Next, a new `User` object is instantiated, and the input arguments to the method are used to populate its various fields.

Next, we come to our first usage of Hibernate. First, we call the `getHibernateSession()` of the `Utils` class, which we looked at earlier. With that session in hand, we next call `beginTransaction()` on it, since everything we do should be in the context of a transaction. We then simply have to call the `save()` method, passing it the `User` object, and finally call `commit()` on the transaction, which we obtain via a call to `session.getTransaction()`. That's it! Notice there is no SQL anywhere to be found, and we didn't have to actually tell Hibernate specifically what to do. It knows how to map the `User` object to the underlying database tables and knows how to create the correct SQL statements to accomplish this (remember, Hibernate is, of course, using JDBC at the end of the day). Very nifty, and if this is your first encounter with Hibernate, you should now be suitably impressed!

Lastly, you can see the call to `Utils.callUpdateData()`, which is the reverse Ajax method we saw earlier, so at this point all users currently using Timekeeper will have their displays updated with the change to the user list.

updateUser(): Or, Why Is Your Wife Still Trying to Change You?!?

The next method is similar, but different, and is for updating existing users:

```
@RemoteMethod
public void updateUser(final String inFieldToUpdate, final long inID,
  final String inNewValue) throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("updateUser() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.debug("updateUser() - inFieldToUpdate = " + inFieldToUpdate);
    log.debug("updateUser() - inID = " + inID);
    log.debug("updateUser() - inNewValue = " + inNewValue);
  }
  User user = getUserByID(inID);
  if (inFieldToUpdate.equals("username")) {
    user.setUsername(inNewValue);
  } else if (inFieldToUpdate.equals("password")) {
    user.setPassword(inNewValue);
  } else if (inFieldToUpdate.equals("isAdministrator")) {
    user.setIsAdministrator(Boolean.parseBoolean(inNewValue));
  } else if (inFieldToUpdate.equals("isProjectManager")) {
    user.setIsProjectManager(Boolean.parseBoolean(inNewValue));
```

```
  }
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  session.update(user);
  session.getTransaction().commit();
  Utils.callUpdateData();
  if (log.isTraceEnabled()) {
    log.trace("updateUser() - Exit");
  }

} // End updateUser().
```

In actuality, the only notable difference is the branching where we determine what field is being updated. First, the appropriate `User` object is retrieved via a call to `getUserByID()`. Next, recall that the updates fire `onBlur` of any field, so we are assured that only a single field is being updated at a given time (although multiple Ajax events could in theory be in flight at any time). So, once the `inFieldToUpdate` argument is interrogated, the appropriate field of the `User` object is updated, and then we call `update()` of the `session` object, instead of `save()` as we saw in the previous method, and the user is updated in the database.

We end again with a call to `Utils.callUpdateData()` to get the changes out to all clients, and the method is done.

You may not like the branching here, and it's a reasonable objection (my technical reviewer raised it, as a matter of fact). In a case like this, you might consider simply submitting all the values for the user, and then comparing what came in via request parameter to their current values in the database. In that way, the code can automatically determine what needs to be updated. Alternatively, you could, of course, always update all fields regardless of whether they changed or not (I prefer knowing that what I'm writing actually changed, if for no other reason than accuracy of auditing down the road). If you had a larger object with more fields or an object with fields that might change with some frequency, the "have the code figure out what changed" approach would almost certainly be the better solution. With a small, limited set of fields though, to me, some simple branching is easier code to follow.

---

deleteUser(): What Happens When Your Wife STOPS Trying to Change You!

Next is the method that deletes a user, and it's probably the simplest of the bunch, and certainly the shortest in terms of lines of code:

```
@RemoteMethod
public void deleteUser(final long inID) throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("deleteUser() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.debug("deleteUser() - inID = " + inID);
```

```
  }
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  User user = (User)session.createQuery(
    "from User as user where user.id = ?")
    .setLong(0, inID).uniqueResult();
  session.delete(user);
  session.getTransaction().commit();
  Utils.callUpdateData();
  if (log.isTraceEnabled()) {
    log.trace("deleteUser() - Exit");
  }

} // End deleteUser().
```

Here is our first encounter with HQL, Hibernate's native query language, and it's used to retrieve the appropriate `User` object. It actually doesn't look a **whole** lot different than normal SQL does, the one interesting difference being that in regular SQL you'd name a table to retrieve from, here you instead specify the object type, `User` in this case. So this statement is basically equivalent to the SQL statement `select * from users where id='XX'`, assuming `XX` was replaced with the user ID and that there was in fact a table named `users` in the database. You can also see here the chaining of method calls, specifically using the `setLong()` method of the `Query` object returned by `session.createQuery()` to insert the `inID` value into the query. The `uniqueResult()` call is simply telling Hibernate to execute the query and return the expected single result, in other words, the `User` object that the ID applies to. Then, we simply call `session.delete()`, passing it the `User` object, and commit the transaction, and the user is at that point deleted. A final call to `Utils.callUpdateData()` is all that remains, and the method's work is complete.

listUsers(): It's Always Nice to Know Who Your Friends Are

The next method one finds as one flips through this class is the `listUsers()` method, and looking at it, I realize I spoke too soon earlier in saying the `deleteUser()` is the simplest because clearly `listUsers` is, as you can see for yourself:

```
@RemoteMethod
public List listUsers() throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("listUsers() - Entry");
  }
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  List usersList = session.createQuery(
    "from User as user order by user.username").list();
  session.getTransaction().commit();
```

```
  if (log.isDebugEnabled()) {
    log.debug("listUsers() - usersList = " + usersList);
  }
  if (log.isTraceEnabled()) {
    log.trace("listUsers() - Exit");
  }
  return usersList;

} // End listUsers().
```

Here again is another usage of HQL, and you can also see a new `list()` method in use of the `Query` object. This, as you'd expect, returns a `List` of `User` objects. The HQL statement is equivalent to `select * from users order by username`, so as you can plainly see, HQL isn't much of a change from plain ole SQL.

getUserByName(): Because Your Dog Doesn't Understand "Dog" Any More Than "Scraps"

Coming up next is another simple method, `getUserByName()`, which we've previously seen called upon by other code:

```
@RemoteMethod
public User getUserByName(final String inUsername) throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("getUserByName() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.trace("getUserByName() - inUsername = " + inUsername);
  }
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  User user = (User)session.createQuery(
    "from User as user where user.username = ?")
    .setString(0, inUsername).uniqueResult();
  session.getTransaction().commit();
  if (log.isDebugEnabled()) {
    log.debug("getUserByName() - user = " + user);
  }
  if (log.isTraceEnabled()) {
    log.trace("getUserByName() - Exit");
  }
  return user;

} // End getUserByName().
```

Another simple HQL query is all it takes to get the job done. This time you can see the `setString()` method in use. As you can guess, the `Query` object returned by `createQuery()` has quite a few setter methods for various data types, enough to suit all your likely needs.

getUserByID(): A Rose, by Any Other Name

Only a single method remains to be examined, and it's the `getUserByID()` method, which we've also seen called upon. If you're guessing that it's virtually identical to `getUserByName()`, give yourself a gold star, because you're absolutely correct!

```
@RemoteMethod
public User getUserByID(final Long inID) throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("getUserByID() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.trace("getUserByID() - inID = " + inID);
  }
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  User user = (User)session.createQuery(
    "from User as user where user.id = ?")
    .setLong(0, inID).uniqueResult();
  session.getTransaction().commit();
  if (log.isDebugEnabled()) {
    log.debug("getUserByID() - user = " + user);
  }
  if (log.isTraceEnabled()) {
    log.trace("getUserByID() - Exit");
  }
  return user;

} // End getUserByID().
```

In fact, except for the just slightly different HQL query, and using `setLong()` instead of `setString()`, these two methods **are** identical.

Now that we've finished looking at all the code pertaining to users, let's flip over to the code pertaining to projects, beginning with the `Project` VO class itself.

Project.java

Like the `User` class before it, the `Project` class is a simple VO that describes a project to Timekeeper. As a simple VO, this class is just a collection of private fields with a getter and setter for each, so looking at the source code wouldn't prove all that revealing, and thus I've left it out. However, Table 9-3 summarizes the fields found in this class and gives you a good idea of what information this VO conveys.

Table 9-3. *The Fields Contained in the* `Project` *Class*

Data Type	Name	Description
Long	`id`	Unique ID of the project. This is automatically populated by Hibernate.
String	`name`	The name of the project.
Long	`projectManager`	The project manager of the project. This field maps to the `id` field of a `User` object.
Integer	`allocatedHours`	The number of hours allocated to this project.
Integer	`bookedHours`	The number of actual hours booked to this project so far.
Date	`targetDate`	The date this project is being targeted to be complete.
String	`usersAssigned`	A comma-separated list of user IDs that are assigned to this project.

ProjectDAO.java

The `ProjectDAO`, which supplies all the database functions dealing with projects, is shown in UML class diagram form in Figure 9-14.

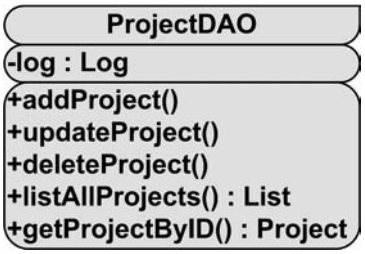


Figure 9-14. *UML class diagram of the* `ProjectDAO` *class*

As was the case when we looked at the client-side code, there's actually not much benefit to reviewing the code for this class here, because it is, to a very high degree, similar to the `UserDAO` class that we looked at earlier. Once again, simply replace the word "user" with "project" everywhere, and you largely get the picture. Yes, the HQL queries are different, and there are a few extra methods in the `UserDAO` class that aren't in the `ProjectDAO` class, but other than that, they are very much the same.

There is, however, one difference worth nothing, and that's in the `listAllProjects()` method, which conceptually corresponds to the `listUsers()` method of the `UserDAO` (that method could have been named `listAllUsers()` and internally it would be the same). Here's the pertinent code:

```
@RemoteMethod
@SuppressWarnings("unchecked")
public List listAllProjects() throws Exception {
```

```
  if (log.isTraceEnabled()) {
    log.trace("listAllProjects() - Entry");
  }

  // Get list of projects.
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  List<Project> projectsList = session.createQuery(
    "from Project as project order by project.name").list();
  session.getTransaction().commit();

  // Now for each, calculate the number of booked hours.
  for (Project p : projectsList) {
    p.setBookedHours(new TimesheetDAO().getBookedTimeForProject(p.getId()));
  }

  if (log.isDebugEnabled()) {
    log.debug("listAllProjects() - projectsList = " + projectsList);
  }
  if (log.isTraceEnabled()) {
    log.trace("listAllProjects() - Exit");
  }
  return projectsList;

} // End listAllProjects().
```

The thing worth noting here is that after the query is done to get the list of projects, similar to how `listUsers()` gets the list of users, the code then uses a `TimesheetDAO` (which is coming up shortly) to calculate the number of hours booked for the project. It's interesting that the number of booked hours isn't stored directly as part of the project itself; it's instead a dynamically calculated value. This is done to abstract out the `TimesheetItem` objects from the `Project` objects they are (loosely) attached to.

With users and projects out of the way, only a single piece is missing, and that's dealing with timesheets and timesheet items. Here we go, closing in on the conclusion!

TimesheetItem.java

Like the `Project` and `User` VO classes we looked at a short while ago, the `TimesheetItem` class is again a plain old VO. This time, it describes a single entry a user makes for a given project on a given day stating the number of hours booked to that project. When you look at the timesheet portion of the home page, each text box, or more precisely the value you enter in it, is represented in Timekeeper by a `TimesheetItem` instance.

As before, the source for this class isn't shown here, but Table 9-4 gives you the overview of the fields it contains.

Table 9-4. *The Fields Contained in the* `Project` *Class*

Data Type	Name	Description
Long	`id`	Unique ID of the item. This is automatically populated by Hibernate.
Long	`userID`	The ID of the user that this item is stored for. This maps to the `id` field of a `User` object.
Long	`projectID`	The ID of the project that this item applies to. This maps to the `id` field of a `Project` object.
Date	`reportDate`	The date the hours represented by this instance applies to.
Integer	`hours`	The number of hours for this project on this day booked by this user.

TimesheetDAO.java

The `TimesheetDAO` class, which can be seen in UML class diagram form in Figure 9-15, has a lot in common with the `UserDAO` and `ProjectDAO` classes, in fact so much so that all but one of the methods probably doesn't need to be reviewed here.

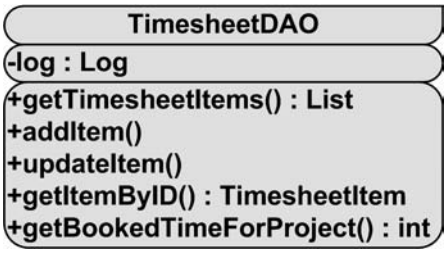


Figure 9-15. *UML class diagram of the* `TimesheetDAO` *class*

The `getTimesheetItems()` method is akin to the `listAllProjects()` and `listUsers()` methods from the `ProjectDAO` and `UserDAO` classes, respectively. The `addItem()` and `updateItem()` methods are just like `addUser()`/`addProject()` and `updateUser()`/`updateProject()`. The only difference is in the HQL statements in play, and of course the objects being manipulated (`TimesheetItem` objects instead of `User` and `Project` objects). The `getItemByID()` method is just like the `getProjectByID()` and the `getUserByID()` methods.

Only one method is actually unique and deserves looking at, and that's the `getBookedTimeForProject()` method, shown here:

```
@RemoteMethod
@SuppressWarnings("unchecked")
public int getBookedTimeForProject(final Long inProjectID) throws Exception {

  if (log.isTraceEnabled()) {
    log.trace("getBookedTimeForProject() - Entry");
  }
  if (log.isDebugEnabled()) {
    log.debug("getBookedTimeForProject() - inProjectID = " + inProjectID);
  }
```

```
  Session session = Utils.getHibernateSession();
  session.beginTransaction();
  List<TimesheetItem> itemList = session.createQuery(
    "from TimesheetItem as item where item.projectID = ?")
    .setLong(0, inProjectID).list();
  int bookedTime = 0;
  for (TimesheetItem ti : itemList) {
    bookedTime += ti.getHours().intValue();
  }
  if (log.isDebugEnabled()) {
    log.debug("getBookedTimeForProject() - bookedTime = " + bookedTime);
  }
  if (log.isTraceEnabled()) {
    log.trace("getBookedTimeForProject() - Exit");
  }
  return bookedTime;

} // End getBookedTimeForProject().
```

First, a `List` `TimesheetItem` object is retrieved for the given project ID. Then, that `List` is iterated over, and we accumulate the value of the `hours` field, via the `getHours()` call, in the variable `bookedTime`. That value is returned and is the total number of hours booked to the specified project across all users and all dates.

And with that, we've completed dissection of the Timekeeper application! I think you'll agree that the combination of DWR, Hibernate, and Ext JS is a potent one that offers simplified coding, and a lot less of it, and yet provides a tremendous amount of power to make a pretty decent little Web 2.0 application.

Suggested Exercises

As usual, there are a number of things you could do to make this application that much better and give you more experience with the technologies employed in it. Here are a few suggestions:

- Whenever changes to the home page are present as a result of the Comet request, highlight them using a Yellow Fade Effect. Ext JS provides this feature, and DWR may even (at the time of this writing that feature wasn't in the official release, but may well be by the time you read this, so a quick DWR upgrade should make it available to you, as well as give you experience upgrading DWR). How exactly you determine when something has changed and what particular row of what table to highlight is up to you to figure out!

- Add timesheet approval/rejection capabilities. This is one of those things that gets left on the cutting room floor, so to speak, due to time constraints. As I had envisioned it, there would be an additional `isManager` field in the `User` object and a manager field that would tie a user to a given manager. Then, timesheets would be submitted by users when complete and would appear for their manager in an approval queue. The manager could then view the timesheet and either approve or reject it. There would also be an archival capability so that when a timesheet is submitted and approved, it gets

archived, but can be recalled by a manager (or the submitting user) later. Going along with all this was going to be an automated process that would alert a user, and his or her manager, that a timesheet should have been submitted. As you can see, the plans were pretty grandiose, and it's no wonder I had to drop this: it's no small task to implement this! Still, if you undertake this challenge, I think you'll find the exercise extremely challenging and yet rewarding.

- Use Ext JS's grid support in place of all the tables in the application. I purposely left this one undone so I could offer it as a suggestion. If you're interested in gaining experience with Ext JS, this is a great place to start. Ext JS offers plain grids, which would work nicely for the project summary display on the home page, as well as grids with editable components, which should do very nicely for all the rest.

Summary

In this chapter, you developed your DWR chops a little further and got to play with some more utility functions, as well as some more Comet-based reverse Ajax. You saw how annotations and XML-based configuration can be mixed and matched as you see fit. You got some first-hand experience with the wonderful Ext JS widget library and saw how it can, in conjunction with DWR, make creating a really cool-looking Web 2.0 application almost child's play. You saw how Hibernate as your data access level can free you from writing a bunch of mundane JDBC code. Finally, put all together, you learned how all the pieces come together to create a fairly useful little application.

Let me take this opportunity to say, now as we come to the end of our journey, that I hope you've enjoyed reading this book as much as I enjoyed writing it. I think you'll agree that DWR is one of the most powerful and yet simple-to-use tools available to the modern-day Ajax developer (and is there any other kind of Ajax developer?). In the process, you got to see a number of other top-notch supporting players in the form of other libraries and toolkits that, when combined with DWR, allow you to take your web application development to the next level. Thanks for reading, and now get out there with your new skills and create the next great Google Maps, Flickr, Facebook, or YouTube! (And hey, don't forget about little ol' me when you're a bazillionaire!)

Index

Numbers and Symbols

$ notation, 123
3270 mainframes, 4–5

A

About dialog box, 313
accessibility
  with Ajax, 23–24
  importance of, 12
accessor (or getter) method, 153
Acegi Security, 203
Actions, Struts, 120
ActionScript, 37
activeReverseAjaxEnabled parameter, 69, 115, 466
ActiveX, 31
Adaptive Path, 14
addArticle() method, 217–218, 244–245
addArticleHistory() method, 251–253
addBeanPropertySetter rule, 387
addComment() method, 222, 252
addGroupToPortal() method, 396
addItem() method, 292
addOptions() method, 92
addProject() method, 504
addQuartzJob() method, 413–415
addReportToFavorites() method, 370, 393
addReportToPortal() method, 379, 400
addReportToSchedule() method, 374, 412–413
Address Book contacts, 148
AddressBookManager class, 168–173
addRow() grid method, 301
addRows() method, 92, 154–155
addScript() method, 509
addUser() method, 492–493, 511–512
Administer Projects dialog box, 495, 499–500
Administer Users dialog box, 489
Ajax
  alternatives to, 36–38
  disadvantages of, 23–24
  functions, 151–153
  introduction to, 14–18
  markup, 29–30
  as paradigm shift, 18–21
  request/response process, 102–105
  reverse. *See* reverse Ajax
Ajax libraries, 33–35
Ajax-based applications
  advantages of, 18–19
  examples, 14–16, 25–33
alert() box, 23, 151
AllArticlesIndex.ftl template, 223–224
allowGetForSafariButMakeForgeryEasier parameter, 68
<allow> section, 76–79
allowScriptTagRemoting parameter, 68
annotation-based configuration, 466
annotations, 122–125
  conflicting, 466
  DWR, 420–421
  with Spring, 351
Ant build script, creating, 53–54
ANT_HOME, 51
Apache Ant 1.7.0, 50
Apache Derby, 190, 194–195, 199
Apache TomCat 6.0.13, 50
Apache web server, 8
appConfig.xml file, 347–348, 385
appConfigured variable, 142
Appear effect, 338
application servers, 50, 113
applications
  Ajax-based, 14–16, 18–19, 25–33
  development history, 3
  distributed, 7
  fat-client, 6–7

PC era, 5–7
web-based, 7–11
See also specific applications
APT, 44
Article class, 233–234
articleClicked() method, 218
ArticleComment class, 234
ArticleComments.ftl template, 224–225
ArticleDAO class, 209, 243–257
ArticleHistory.ftl template, 225–226
ArticleHistoryItem class, 234–235
async option, 90
Asynchronous JavaScript and XML (Ajax). *See* Ajax
attachToObject() method, 291
<auth> element, 77, 100, 204
<auth-method> element, 202

B

bean converters, 76, 79, 124, 277
beanName parameter, 118
beans, setting on remote objects, 83–87
Beehive framework, 121
beginTransaction() method, 512
Berners-Lee, Tim, 21
bookmarking, 24
bugs, 101
build.xml file, 199
build_libs directory, 199
built-in converters, 75–76, 79
buttons, hiding, 146
byId() function, 93

C

calculateProjectStatus() method, 497, 504–505
call batching, 92, 487
call syntax, 81, 83
callback attribute, 82
callback functions, 32–33, 64
extending data passing to, 87–88
for effects, 339
with metadata object, 82
passing to proxy stub, 81–82
callback parameter, 120
callUpdateData() method, 509
Cascading Style Sheets (CSS), 13. *See also* styles.css file
ccMail, 6
cellFuncs arrays, 374, 497–499
CGI. *See* Common Gateway Interface
checkboxNumber variable, 142, 153
checkForWin() method
GameCore class, 444
InMemoria class, 439–440
classes
remote access to, 96–98, 420–421
See also specific classes
classes directory, 198
classes init parameter, 123, 466
classic Web, 10–14
CLASSPATH environment variable, 51
clearAll() method, 301
click handlers, 294
client, interacting with DWR on, 81–88
client-side code
DWiki application, 205–229
DWikiClass.js, 215–223
index.jsp, 206–210
login.jsp, 210–211
loginOK.jsp, 212–213
RolloversClass.js, 213–214
styles.css, 206
Fileman application, 278–314
downloadFile.jsp, 288–289
fileman.js, 289–314
index.jsp, 282, 285–286
login.htm, 280–281
loginBad.htm, 281–282
styles.css, 278–280
uploadFile.jsp, 286–288
InMemoria game, 426–441
howToPlay.txt, 430
index.jsp, 427–429
InMemoria.js, 430–441
styles.css, 426–427
InstaMail application, 134–163
index.jsp, 137–141
styles.css, 135–137
script.js, 141–163
RePortal application, 352–385
index.jsp, 359–365
lightbox.css, 353–359

lightbox.js, 353–359
RePortalClass.js, 365–385
styles.css, 352
Timekeeper, 471–507
index.jsp, 473–479
styles.css, 471–473
Timekeeper.js, 479–507
client-side scripting, 9, 11, 13
close() method, 187
closeItem() method, 300
closures, 87–88
code
efficiency of, 509
See also client-side code; server-side code
code monkey, 25
collapseSection() method, 368
collection converter, 76
color blindness, 23
com directory, 199
Comet technique, 112–116
commas, in file names, 302
commentsClicked() method, 221
commit() method, 512
Common Gateway Interface (CGI), 8
Commons Digester, 270, 387–388
Commons FileUpload, 269, 287
Commons IO, 268–269, 324–325
complexity, 24
concurrency issues, with wikis, 190
Config class, 196, 230, 385–388
Config.getDataSource() method, 407
Config.java file, 230
configuration, with annotations, 122–125
configuration files
DWiki application, 199–205
dwiki.properties, 204–205
dwr.xml, 203–204
web.xml, 200–203
Fileman application, 273–278
dwr.xml, 276–278
web.xml, 274–276
InMemoria game, 424–426
web.xml, 424–426
InstaMail application, 132–134
RePortal application, 346–351
appConfig.xml, 347–348
dwr.xml, 348–349
spring-beans.xml, 349–351
web.xml, 346–347
Timekeeper application, 465–471
dwr.xml, 466–467
hibernate.cfg.xml, 467–468
Project.hbm.xml, 469–470
TimesheetItem.hbm.xml, 470–471
User.hbm.xml, 470
web.xml, 465–466
confirmFlip() method, 451
connection.pool_size property, 468
constructor injection, 331
container-managed security, 199–200, 203, 273
Content-Disposition header, 289, 312
contextInitialized() method, 388
ContextListener class, 388–389
converter attribute, 124
<convert> element, 77–79
converters, 75
built-in DWR, 75–76
custom, 76
copying files, 303–304
copyMoveFile() method, 304, 324–325
Coulton, Jonathan, 25
createDirectory() method, 308, 321–322
<create> element, 77, 100
createFile() method, 307, 323
creator attribute, 77
creators, 75
built-in DWR, 75–76
custom, 76, 422
cross-browser issues, 297
cross-domain Ajax calls, 68
crossDomainSessionSecurity parameter, 68
css directory, 197, 270, 278
cssBody class, 279, 473
cssDirectories class, 280
cssDivider class, 280
cssFullScreenDiv class, 280
cssHeader class, 473
cssLightboxBlocker class, 355
cssLightboxPopup class, 356
cssMenu class, 279
cssOuter class, 473
cssSectionDivider class, 352
cssSource class, 473

cssTable class, 473
cssTableAltRow class, 473
cssTableHeader class, 473
cssTableRow class, 473
cssToolbar class, 279
currentView variable, 142, 157
current_session_context_class property, 468
custom attributes, 143
custom converters, 76
custom creators, 76, 422
cutting files, 303–304
cycleOpponentExplosion() method, 440
cyclePlayerExplosion() method, 437–439

D

data converters, 76
data parameter, 153
data passing, extending to callbacks, 87–88
data sets, 4
data sources, listing, 407–408
Data Transfer Objects (DTOs), 134
data variable, 162
database application. *See* RePortal application
DatabaseWorker class, 232, 237–244, 389, 409
DataIntegrityViolationException, 197, 397, 401
dataSource elements, 348
DataSourceDescriptor class, 388, 390
datastreams, 5
@DataTransferObject annotation, 124
DataVision, 333–335, 407
debug parameter, 68
deleteContact() function, 160
<delete> task, 54
deleteFile() method, 306, 322
deleteJob() method, 415
deleteMessages() method, 158, 187
deleteProject() method, 504
deleteUser() method, 494, 513–514
deny by default security, 96–98
dependency injection (DI), 195, 331–333
Derby, 190, 194–195
development environment, 49–52
DHTML eXtensions (dhtmlx), 262
dhtmlx components, 261–267, 273
  cost of, 262
  using, 266–267
  license terms, 273
dhtmlxCombo component, 262–265
dhtmlxFoldersPane component, 266
dhtmlxGrid component, 262–263
dhtmlXItemObject, 294
dhtmlxMenubar component, 266
dhtmlXMenubarObject, 294
dhtmlXMenuBarPanelObject, 294
dhtmlXMenuItemObject, 294
dhtmlxTabbar component, 262–264
dhtmlxToolbar component, 265–266
dhtmlXToolbarDividerXObject, 292
dhtmlxTree component, 262–263, 300
dhtmlxTreeGrid component, 262–264
dhtmlXTreeObject, 290
dhtmlxVault component, 262, 265
dialect property, 468
Digester, 387–388
Direct Web Remoting (DWR). *See* DWR
direction effect, 339
directories
  creating, 308–309, 321–322
  deleting, 305–306
  jumping to parent, 306–307
  listing, 320
Directory Opus, 261
directory tree, 290
directoryClicked() method, 290, 300–303, 305, 309
directoryExpanded() method, 290, 299–300, 303
DirectoryVO class, 316–317, 319
DirectoryWalker class, 268
display properties, abstracting out, 146
display style attribute, 357
displayGroupInfo() method, 383
displayUserInfo() method, 384
distributed applications, 7
 elements, 29–30, 33, 141
divFileEditor, 285
divFileUpload, 286
divPleaseWait, 146–147
DLL Hell, 7
doAbout() method, 313
doCopyCut() method, 303–304
Document Object Model (DOM), 23

doDelete() method, 157–159, 305–306
doDownload() method, 311–312
doEditFile() method, 309–310
Dojo toolkit, 35, 112, 268
doMath() function, 64
doMathCallback() function, 64
DomHelper.append() method, 489
doNewDirectory() method, 308–309
doNewFile() method, 307
doPaste() method, 304–305
doPrintDirectoryContents() method, 313–314
doSaveFile() method, 310
doUpload() method, 312–313
doUpOneLevel() method, 306
doUsing() method, 298
downloadFile.jsp file, 288–289, 311–312
drop executions, 241
drop shadows, 356
dumb terminals, 5
duration effect, 339
DWiki application
  content storage, 194–195
  adding comments, 222
  client-side code, 205–229
    DWikiClass.js, 215–223
    index.jsp, 206–210
    login.jsp, 210–211
    loginOK.jsp, 212–213
    RolloversClass.js, 213–214
    styles.css, 206
  configuration files, 199–205
    dwiki.properties, 204–205
    dwr.xml, 203–204
    web.xml, 200–203
  content storage, 194
  database schema, 238–239
  directory structure, 197–199
  features and functionality, 190
  FreeMarker templates, 223–229
  front page, 205
  login page, 212
  modes, 215
  security configuration, 199–200
  server-side code, 230–257
    Article.java, 233–234
    ArticleComment.java, 234
    ArticleDAO.java, 243–257
    ArticleHistoryItem.java, 234–235
    Config.java, 230
    DatabaseWorker.java, 237–242
    DWikiContextListener.java, 230–232
    Freemarker class, 235–237
  suggested exercises, 257–258
  tools for, 190–197
dwiki.properties file, 204–205
DWikiClass class, 215–223
DWikiContextListener class, 230–232, 237, 240
DWikiHelp.ftl template, 227
dwl.xml file, configuring DWR, 70–81
DWR (Direct Web Remoting)
  adding to webapp, 61–65
  advantages of, 46
  annotations, 420–421
  architecture, 47–48
  Beehive and, 121
  call syntax, 81–83
  with classic Struts, 120–121
  configuring
    dwr.xml file, 70–81
    engine.js file, 90–92
    using annotations, 122–125
    web.xml, 67–70
  converters, 75–79
  creators, 75–76
  development environment, 49–52
  error handling, 101–107
  Hibernate and, 122
  integrating with libraries and frameworks, 117–122
  interacting with on server, 88–90
  interacting with on client, 81–88
  introduction to, 39–40
  JavaScript file generated by, 152
  JSF support in, 119
  overview, 45–48
  reasons to choose, 43–44
  request flow, 49
  security, 95–101
    deny by default approach, 96–98
    J2EE, 98–101
  Spring support in, 118
  test/debug page, 65–67

use of, in InstaMail, 153–155
with WebWorks/Struts 2, 119–120
DWR file manager, 259–260. *See also* Fileman application
DWR initialization code, 294
DWR mailing list, 46
DWR servlet, 133, 138
dwr.jar file, 61
dwr.util() function, 93
dwr.util.addOptions() function, 370, 378
dwr.util.addRows() function, 374, 492, 497, 499
dwr.util.getText() function, 383
dwr.util.getValue() function, 370
dwr.util.removeAllRows() function, 373
dwr.util.setValue() function, 220, 366, 500
dwr.war file, 61
dwr.xml file, 65, 70–81
<allow> section, 76–79
DWiki application, 203–204
Fileman application, 276–278
<init> section, 76
InstaMail application, 133–134
multiple, 98–100
RePortal application, 348–349
<signatures> section, 79–81
Timekeeper application, 466–467
dwr/interface/MathDelegate.js file, 64
DWRActionUtil object, 120
DWRActionUtil.js, 120
DWREngine object, 90–92
DWRServlet class, 48, 62, 64
init parameters, 68–70
securing, 98–100
DWRUtil object, 153–155

E

e-mail applications, 129. *See also* InstaMail application
early closing mode, 113
editClicked() method, 218–219
editContact() method, 148
editFile() method, 310, 325
effects, 337–340
efficiency, 509
ejb3 creator, 75
enableAutoHeight() method, 291
enableButtons() method, 148
endBatch() call, 92
engine.js file, 64, 90–92, 285
Enterprise JavaBeans (EJBs), 19
enum converter, 76
error handling, 308
in DWR applications, 101–107
exceptions, 102, 105–107, 392
mechanics of, 105–106
response errors, 102–105
warnings, 102
errorHandler element, 82, 90
eval() function, 143, 214
eventHandler() function, 47
exception classes, 197
exception handlers, global, 295, 308
exception handling, 102, 105–107, 392
exceptionHandler() method, 296
exceptions, for flow control, 167
<exclude> element, 77
execute() method, 121, 408–409
executeQuery() method, 242, 247
executeUpdate() method, 242
execution context, creation of, 88
expandLinks() method, 247–248
expandSection() method, 367
Ext JS, 461–462, 473, 477–478

F

fat clients, 3–5
fat-client applications, 6–7
favorites functions, 369–372
favoritesDisplayReportInfo() method, 371
FavoritesWorker class, 390–394
FavoritesWorker.addReportToFavorites() method, 370
FavoritesWorker.getFavoritesForUser() method, 369
File class, 324
file manager application. *See* Fileman application
file managers, options for, 260–261
file uploads, 269
File.delete() method, 187
FileItemFactory, 287
Fileman application, 270
client-side code, 278–314

downloadFile.jsp, 288–289
fileman. js, 289–314
index.jsp, 282–286
login.htm, 280–281
loginBad.htm, 281–282
styles.css, 278–280
uploadFile.jsp, 286–288
configuration files, 273–278
dwr.xml, 276–278
web.xml file, 274–276
container-managed security configuration, 273
directory structure, 270–271
interface, 272–273
security features, 276
server-side code, 314–326
DirectoryVO.java, 316–317
FileSystemFunctions.java, 317–326
FileVO.java, 315–316
suggested exercises, 326
Fileman class, 289–314
directoryClicked() method, 300–301
directoryExpanded() method, 299–300
doAbout() method, 313
doCopyCut() method, 303–304
doDelete() method, 305–306
doDownload() method, 311–312
doEditFile() method, 309–310
doNewDirectory() method, 308–309
doNewFile() method, 307
doPaste() method, 304–305
doPrintDirectoryContent() method, 313–314
doSaveFile() method, 311
doUpload() method, 312–313
doUpOneLevel() method, 306–307
exceptionHandler() method, 296
filenameChange() method, 291, 302–303
getContentAreaHeight() method, 296–297
getFullPath() method, 298–299
init() method, 290–296
menubarButtonClick() method, 297–298
onResize() method, 296
private membrs, 289
toolbarButtonClick() method, 298
filenameChanged() method, 291, 302–303
FilenameUtils class, 288
files
copying, 303–304, 324–325
creating, 307, 323
cutting, 303–304
deleting, 305–306, 322
downloading, 311–312
editing, 309–310, 325
listing, 319–320
pasting, 304–305, 324–325
printing contents of, 313–314
renaming, 302–303, 322–323
saving, 325–326
uploading, 312–313
FileSystemFunctions class, 277, 285, 295, 300–304, 317–326
copyMoveFile() method, 324–325
createDirectory() method, 308, 321–322
createFile() method, 307, 323
deleteFile() method, 306, 322
editFile() method, 310, 325
handleDirectory() method, 317–319
listDirectories() method, 320
listFiles() method, 314, 319–320
listRoots() method, 320–321
renameFile() method, 322–323
saveFile() method, 311, 325–326
FileUtils class, 268
FileVO class, 301, 315–316
Flash, 36–37
Flex, 37–38
float-overs, 136–137, 146–147
FORM auth method, 202
<form-login-page> element, 202
formatter function, 369
forwardToString() method, 444
fps effect, 339
frames, 10, 13
frameworks
Beehive, 121
classic Struts, 120–121
Hibernate, 122
integrating DWR with, 118–120
JSF, 119
Spring, 118
WebWork/Struts 2, 119–120
FreeCommander, 261
Freemarker class, 191–194, 232, 235–237

FreeMarker templates, 190, 223–229
AllArticlesIndex.ftl, 223–224
ArticleComments.ftl, 224–225
ArticleHistory.ftl, 225–226
DWikiHelp.ftl, 227
newArticle.ftl, 227
nonexistentArticle.ftl, 227
Search.ftl, 228–229
SearchResults.ftl, 229
from effect, 339
full-stack framework, Spring as, 195
full-streaming mode, 113
functions
execution context, 88
in JavaScript, 145
See also specific functions

G

game application. *See* InMemoria
GameCore class, 441–445
Garrett, Jesse James, 14
generateGrid(), 442
generateGrid() method, 442–443
GET method, 32
Getahead, 39, 47
getAllArticles() method, 246–249
getArticle() method, 215–216, 245–248, 255
getArticleComments() method, 251
getArticleHistory() method, 221, 250–251
getASession() method, 509
getBookedTimeForProject() method, 519
getBookedTimeForProjectByDate() method, 505–506
getCheckbox() method, 153
getContentAreaHeight() method, 291, 296–297
getDataSourceList() method, 407–408
getFavoritesForUser() method, 391, 393, 396
getFullPath() method, 298–299, 306, 310
getGroupLists() method, 395
getHibernateSession() method, 509, 512
getInboxContents() method, 152, 183
getInstance() method, 447
getJdbcTemplate() method, 239
getName() method, 288
getOptions() method, 47
getParentId() method, 307
getReportParameters() method, 417
getReportsList() method, 399
getScheduledReportsList() method, 366, 412–415
getSentMessagesContents() method, 183
getStaticArticle() method, 249–250, 257
getText() method, 93
getTime() method, 485
getTimesheetItems() method, 505, 519
getType() method, 447
getUserByID() method, 484, 494, 516
getUserByName() method, 515
getUserData() method, 301, 303, 308
getUserPrincipal().getName() method, 255
getValue() method, 93
getXXXX functions, 146
global exception handlers, 295, 308
global variables, 142–143
Gmail, 21–22
Goodwin, Mark, 39
Google Maps, 19–20
Google Reader, 22
Google Web Toolkit (GWT), 38
gotoAddressBook() function, 159
gotoComposeMessage() function, 150
gotoComposeReply() function, 150
gotoHelp() function, 151
gotoInbox() function, 151–152, 155, 159
gotoOptions() function, 160
gotoSentMessage() function, 155, 158
gotoViewMessage() function, 156–157
graphics, 13
green screens, 5
group functions, 381–383
GroupDescriptor class, 394
groups, deleting, 397
GroupWorker class, 394–397

H

hackers, 95
handleDirectory() method, 317–319
hasChildren field, 300, 319
hbm2ddl.auto property, 468
headers option, 90
hiberante.cfg.xml file, 467–468

Hibernate, 122, 459–460, 468–471, 512
  Project class definition file, 469–470
  TimesheetItem class definition file, 470–471
  User class definition file, 470
Hibernate Query Language (HQL), 459
hibernate2 converter, 122
hibernate3 converter, 122
hideAllLayers() function, 146
hidePleaseWait() function, 148
historyClicked() function, 220
Hotmail, 21
hover state, 136
howToPlay() method, 440, 444
howToPlay.txt file, 430
HSQLDB database, 458–459
HTML 4.01, 13
HTTP errors, 102
HTTP methods, 32
HTTP requests, 32, 102
HTTP-based file uploads, 269
httpMethod option, 90
HttpServletRequest, 401

I

id attributes, 64
id parameter, 120
IDE, 36, 52
ignoreLastModified parameter, 69
image preloads, 142
image rollovers, 142
img directory, Fileman, 270
imgOut() function, 214
imgOver() function, 214
INBOX folder, 183
<include> method, 277
inDepth parameter, 318
index.jsp file, 55–56, 63–64
  DWiki application, 206, 209–210
  Fileman application, 282–286
  InMemoria game, 427–429
  InstaMail application, 137–141
  RePortal application, 359–365
  Timekeeper application, 473–479
indexOf() method, 400
inDirectory parameter, 319
init parameters, 115, 123
<init> section, 76
init() method
  DWikiClass class, 209, 215, 232, 237
  Fileman class, 285, 290–296
  InMemoria class, 431–434
  InstaMail class, 138, 140
  RePortalClass.js file, 365–367
  Timekeeper class, 477, 483–485
initUI() method, 480–482
InMemoria (game application)
  application requirements and goals, 419–420
  client-side code, 426–441
    howToPlay.txt, 430
    index.jsp, 427–429
    InMemoria.js, 430–441
    styles.css, 426–427
  configuration files, 424–426
    web.xml, 424–426
  directory structure, 423–424
  DWR annotations, 420–421
  reverse Ajax, 421–422
  server-side code, 441–456
    GameCore class, 441–445
    Opponent class, 447–456
    OpponentCreator class, 445, 447
  startup screen, 423
  suggested exercises, 456
InMemoria class, 430–441
  checkForWin() method, 439–440
  cyclePlayerExplosion() method, 438–439
  data fields, 430–431
  howToPlay() method, 440
  init() method, 431–434
  opponentWon() method, 453
  startGame() method, 434–435
  tileClick() method, 435–438
inMemoria.init() method, 428
InMemoria.js file, 430–441
InMemoria.opponentFlipTile() method, 450
innerHTML property, 33
innerHTML values, 149
inOperation parameter, 304
inResults parameter, 318
insertNewChild() method, 295, 300

InstaMail application, 129
Address Book, 159–160
AddressBookManager.java, 168–173
client-side code, 134–163
composing messages, 150–151
configuration files, 132–134
configuring options-related code, 160–163
contact management, 159–160
deleting messages, 157–159
directory structure, 130–132
index.jsp, 137–141
MailDeleter.java, 183–187
MailRetriever.java, 177–183
MailSender.java, 173–177
option screen, 161
OptionsManager.java, 163–168
requirements and goals, 129–130
screenshot, 134
script.js, 141–163
sending messages, 159
server-side code, 163–187
styles.css, 135–137
suggested exercises, 187–188
use of DWR in, 153–155
viewing messages, 156–157
integrated development environment (IDE), 36, 52
interface injection, 331
Internet Time, 11
Inversion of Control (IoC), 88–89, 196, 331–333
inWriteHistory flag, 254–255
IOException, 167
isUserAssignedToProject() method, 495–496
isUserInRole() method, 251
items, listing, 319–320
ivFileUpload, 285

J

J2EE security, 98–101, 211
Jakarta Commons, 269–270
Jakarta Commons IO, 268–269
Jakarta Commons Logging, 198
Jakarta Commons web site, 270
Java Web Parts (JWP), 44
JavaMail API, 183, 187
JavaScript, 39
asynchronous nature of, 64
DWR servlet and, 138
functions, 30–32
generated by DWR, 152
libraries, 34–35
prototyping in, 145
javascript attribute, 77
JavaScript Object Notation (JSON), 105
JavaServer Pages (JSPs), 19
JAVA_HOME, 51
JBoss, 459
JDBC, 190, 196
JdbcTemplate class, 196–197, 249
JobDataMap, 409
JobExecutionContext object, 409
jobs, 336
js directory, 197, 270
JSF (JavaServer Faces), 119
JSFCreator, 75, 119
j_security_check servlet, 281

K

KISS principle, 140–141

L

lib directory, 198
libraries
integrating DWR with, 118–119
JavaScript, 34–35
JSF, 119
Spring, 118
licenses, 262
lightbox.css file, 353–359
lightbox.js file, 353–359
lightboxes, 330, 353–359, 364
list() method, 319
listAllProjects() method, 496, 517
listDirectories() method, 300, 320
listFiles() method, 301, 314, 319–320
listRoots() method, 295, 320–321
listUsers() method, 514–515
location parameter, 118
lockArticleForEditing() method, 219, 254–255
<login-config> element, 202
login.html file, 280–281

login.jsp file, 210–211
loginBad.html file, 281–282
loginOK.jsp, 212–213
logUserIn() method, 385, 403, 476–477, 510–511

M

magic numbers, 148
MailDeleter class, 158, 183–184, 186–187
MailRetriever class, 152, 157, 177–183
MailSender class, 159, 173–177
mainframe applications, 3–5
Manage Projects dialog box, 500–503
Mappr, 36
match attribute, 79
MathDelegate class, 58, 60, 64
mathservlet.java file, 56–58
maxCallCount parameter, 69
maxPollHitsPerSecond parameter, 69
maxWaitingThreads parameter, 69
McCool, Rob, 8
menubar, construction of, 293
menubarButtonClick() method, 297–298
menuClickHandler() method, 485–486
MessageDTO object, 177, 183
metadata objects, 82
methods
  calling, 451
  remote, 277
  securing individual, 100–101
  See also specific methods
Microsoft Windows, 5
MimeMessage object, 177
MinimalistExceptionConverter, 106
<mkdir> task, 54
Model-View-Controller (MVC), 195
MooTools, 35
mouseOver events, 149
mpa converter, 76
multimedia, 9
.mxml file extension, 37
My Favorites section, 361–363

N

n parameter, 289
name attributes, 64
navigation menu, DWiki, 209
NCSA HTTP web server, 8
network traffic, 14
new creator, 75
newArticle.ftl template, 227
newContact() function, 150
nonexistentArticle.ftl template, 227
nonhover state, 136
normalizeIncludesQueryString parameter, 70
null, 32

O

object converter, 76
Object-Relational Mapping (ORM) tools, 459–460
ObjectCreate rule, 388
onBlur event, 492
ONC RPC, 39
onChange events, 31
onClick events, 149
onClick event handler, 157
onLoad event handler, 477
onMouseOver property, 143–146
onReady events, 481
onResize() method, 285, 296
onReturn() method, 93
open() method, 32
Opponent class, InMemoria game, 447–456
OpponentCreator class, InMemoria game, 445–447
opponentFlipTile() method, 450
opponentNoMatch() method, 453
opponentWon() method, 453
OptionsDTO object, 140, 162, 167
OptionsManager class, 163–168
ordered option, 91
org.apache.commons.io, 268
org.apache.commons.io.filefilter, 268
org.apache.commons.io.input, 268
outcomeComplete() method, 452–453
overflow attribute, 357
overridePath parameter, 70

P

p parameter, 289
pageflow creator, 75, 121
<param> elements, 77
param parameter, 120

parameters. *See specific types*
parameters option, 90
params HashMap, 288
parentNode property, 157
parse() method, 287
parseInt() method, 368
path separator character, 285
personal computers (PCs), 5–7
pickTile() method, 454–455
piggybacking, 114–117
Please Wait float-over, 136
POJOs (Plain Old Java Objects), 118
polling technique, 111–112, 116
pollType option, 91
POP3 protocol, 129
POP3 servers, 183
populateList() method, 48
postHook option, 91
postStreamWaitTime parameter, 69
Practical Ajax Projects with Java Technology, 17
Practical JavaScript, DOM Scripting, and Ajax Projects, 17
preHook option, 91
preStreamWaitTime parameter, 69
professional open source projects, 459
Project class, Timekeeper application, 516–517
Project.hbm.xml file, 469–470
ProjectDAO class, Timekeeper application, 517–518
Properties object, 168, 173, 183, 187
prototyped-based languages, 145
Prototype, 35
prototyping, in JavaScript, 145
proxy stub objects, 45, 81–82
push technology, 110

Q

Quartz scheduler, 336–337, 389, 408–409
  adding jobs to, 413–414
  removing reports from, 414–415
  starting, 415
queryForList() method, 196, 242, 249
queryForMap() method, 196
queue effect, 339

R

randomlyCycleTiles() method, 433
read() method, 108–109
readystate code, 33
relational databases, Derby, 194–195
remote classes, 96–98, 420–421
remote method invocation, 39
remote methods, 277
remote objects, setting beans on, 83–87
Remote Procedure Calls (RPCs), 39, 45–46
remote signatures, 89
RemoteClass class, 124
@RemoteMethod annotation, 124, 451
@RemoteProperty annotation, 124
@RemoteProxy annotation, 124, 420–421
removeAllOptions() method, 93
removeAllRows() method, 93, 153
removeGroupFromPortal() method, 397
removeReportFromFavorites() method, 372, 394
removeReportFromPortal() method, 401
removeSection() method, 367
renameFile() method, 303, 322–323
replaceTokens() method, 241–242
replyGetInboxContents() method, 151–152
report functions, 377–381
report portal application. *See* RePortal application
report portals, overview of, 329–330
RePortal application
  client-side code, 352 –385
    index.jsp, 359–365
    lightbox.css, 353–359
    lightbox.js, 353–359
    RePortalClass.js, 365–385
    styles.css, 352
  components
    DataVision, 333–335
    Quartz, 336–337
    script.aculo.us, 337–340
    Spring, 331–333
  configuration files, 346–351
    appConfig.xml, 347–348
    dwr.xml, 348–349
    spring-beans.xml, 349–351
    web.xml, 346–347
  database, 351–352

dependency injection, 331–333
directory structure, 341–342
expanding and collapsing sections, 363–364
introduction to, 329
My Favorites section, 361–363
requirements and goals, 329–330
sample database for, 340–341
screen shot, 342–345
server-side code, 385–416
Config class, 385–388
ContextListener class, 388–389
DatabaseWorker class, 389
DataSourceDescriptor class, 390
FavoritesWorker class, 390–394
GroupDescriptor class, 394
GroupWorker class, 394–397
ReportDescriptor class, 397
ReportRunner class, 404–409
ReportScheduleDescriptor class, 409
ReportSchedulingWorker class, 410–416
ReportWorker class, 398–401
UserDescriptor class, 402
UserWorker class, 402–404
Spring integration, 348–351
suggested exercises, 416–417
RePortal.removeReportFromFavorites() method, 370
RePortalClass.js file, 365–385
favorites functions, 369–372
group functions, 381–383
init() method, 365–367
report functions, 377–381
scheduling functions, 372–377
section functions, 367–368
user functions, 383–385
ReportDescriptor class, 397
ReportRunner class, 404–409
reports
adding, 400
adding to schedule, 412–413
deleting, 401
listing, 399–400, 412
output of, 406
removing from scheduler, 414–415
running, 406–407
viewing scheduled, 415–416
ReportScheduleDescriptor class, 374, 409
ReportSchedulingWorker class, 410, 412–416
ReportSchedulingWorker.addReportToSchedule() method, 376
ReportSchedulingWorker.viewScheduledRun() method, 377
ReportWorker class, 398–401
request handling, 32–33
request.isUserInRole() function, 204
request/response process, 102–105
response errors, 102–105
retrieveContact() method, 173
retrieveMessage() method, 183
retrieveMethod() method, 157
retrieveOptions() method, 162, 167–168
reverse Ajax, 70, 109–117, 452
code of, 115–117
Comet technique, 112–116
early closing mode, 113
full-streaming mode, 113
passive, 114–115
piggybacking, 114–117
polling technique, 111–112, 116
sequence of events in, 110–113
use of, in game application, 448–456
workings of, 421–422
reverseAjax option, 91
Rich Internet Application (RIA), 21
RollerOversClass.js file, 213–214
row striping, 136
rowClass attribute, 143
rowCreator element, 497
rpcType option, 90
run() method, 237, 452
runReport() method, 380, 406–409
runtime problems, 101

S

s parameter, 64
save() method, 512
saveContact() method, 159–160
saveFile() method, 285, 311, 325–326
saveOptions() method, 162–163, 168
saveTimeSheetItem() method, 506–507
scalability issues, with reverse Ajax, 113
scheduleJob() method, 414
scheduling functions, 372–377

scheduling systems, 336–337
scope attribute, 77
screen readers, 23
<script> element, 48
script sites, 34
script.aculo.us library, 35, 337–340
script.js file
  global variables, 142–143
  image preloads, 142
  InstaMails, 141–163
  onMouseOver property, 143–146
scriptaculous.js file, 360
scriptSessionTimeout parameter, 69
scrollTop attribute, 358
search() method, 256–257
Search.ftl template, 228–229
SearchResults.ftl template, 229
section functions, 367–368
security
  container-managed, 199–200, 203, 273
  in DWR, 95–101
    deny by default approach, 96–98
    J2EE, 98–101
  Fileman application, 276
  stack trace elements and, 107
<security-constraint> element, 202
<Security-role> element, 202
security frameworks, 203
security threats, 96
<select> elements, 29–30, 33
selectItem() method, 307
selectRange() method, 93
send() method, 32
sendMessage() method, 159, 177, 183
server
  interacting with DWR on, 88–90
  rendering of pages by, 13–14
server push technique, 111
server-side code
  DWiki, 230–257
    Article.java, 233–234
    ArticleComment.java, 234
    ArticleDAO.java, 243–257
    ArticleHistoryItem.java, 234–235
    Config.java, 230
    DatabaseWorker.java, 237–242
    DWikiContextListener.java, 230–232
    Freemarker class, 235–237
  Fileman application, 314–326
    DirectoryVO.java, 316–317
    FileSystemFunctions.java, 317–326
    FileVO.java, 315–316
  InMemoria game, 441–456
    GameCore class, 441–445
    Opponent class, 447–456
    OpponentCreator class, 445–447
  InstaMail, 163–187
    AddressBookManager class, 168–173
    MailDeleter class, 183–187
    MailRetriever class, 177–183
    MailSender class, 173–177
    OptionsManager class, 163–168
  RePortal, 385–416
    Config class, 385–388
    ContextListener class, 388–389
    DatabaseWorker class, 389
    DataSourceDescriptor class, 390
    FavoritesWorker class, 390–394
    GroupDescriptor class, 394
    GroupWorker class, 394–397
    ReportDescriptor class, 397
    ReportRunner class, 404–409
    ReportScheduleDescriptor class, 409
    ReportSchedulingWorker class, 410–416
    ReportWorker class, 398–401
    UserDescriptor class, 402
    UserWorker class, 402–404
  Timekeeper, 507–520
    Project.java, 516–517
    ProjectDAO.java, 517–518
    TimesheetDAO.java, 519–520
    TimesheetItem.java, 518–519
    User.java, 509–510
    UserDAO.java, 510–516
    Utils.java, 507–509
ServerLoadMonitor parameter, 115
service code, 45
Servlet APIs, 53
servlet containers, spawning threads in, 337
ServletContext, 173
ServletFileUpload, 287
session.createQuery() method, 514

session.delete() method, 514
session.getTransaction() method, 512
sessionCookieName parameter, 69
setChecked() method, 140
setDisabled() method, 140
setHeader() method, 291
setInterval() method, 367
setLong() method, 514
setName() method, 315
SetNext rule, 388
setString() method, 515
setter injection, 331
setter methods, 91
setType() method, 315
setUserData() method, 300–301
setValue() method, 93, 116, 140
setValues() method, 93
setXXXX function, 146
showHideAddSection() method, 368
showLoading() method, 216
showMessage() method, 487–489
showPleaseWait() method, 146–147
showView() method, 148
show_sql property, 468
<signatures> section, 79–81
Simple Mail Transfer Protocol (SMTP), 129
Slashdot, 9
source code, 52
spawning threads, 337
Spring, 118
  annotations with, 351
  configuration, 332
  integration of, in RePortal, 348–351
  IoC, 88, 331–333
  library, 190
  popularity of, 196
Spring beans, 118
spring creator, 75, 118
Spring JDBC, 195–197
spring-beans.xml file, 332, 349–351
SQL statements
  in FavoritesWorker class, 391
  in GroupWorker class, 395
  in ReportRunner class, 405
  in ReportWorker class, 399
  in UserWorker class, 403
  in WorkSchedulingWorker class, 411
SQLExceptions, 197
src directory, 199, 271
stack trace elements, 107
startGame() method
  GameCore class, 441–442
  InMemoria class, 434–435
startPolling() method, 117
startScheduler() method, 415
Struts, classic, 120–121
Struts 2, 119–120
struts creator, 75
style attributes, 141, 146
styles sheets, 146, 206
styles.css file
  DWiki application, 206
  Fileman application, 278–280
  InMemoria game, 426–427
  InstaMail application, 135–137
  RePortal application, 352
  Timekeeper application, 471–473
Sun Java SDK 1.6.0_03, 50
@SuppressWarnings annotation, 392
switchMode() method, 216–219
sync effect, 339

T

tables
  DWR and, 155
  styling, 471–473
TARDIS (Time And Relative Dimensions In Space), 507
template engines, 191–194
templates directory, 197
terminal emulation applications, 3–5
test/debug page, 65–67
textHtmlHandler option, 91
this keyword, 145, 149
this.parentNode call, 149
thread starving, 113
ThreadLocal, 320
threads, spawning in servlet container, 337
tileClick() method, 435–438
Timekeeper (project management application)
  Administer Projects dialog box, 495, 499–500

application requirements and goals, 457–458
client-side code, 471–507
index.jsp, 473–479
styles.css, 471–473
Timekeeper.js, 479–507
configuration files, 465–471
dwr.xml, 466–467
hibernate.cfg.xml, 467–468
Project.hbm.xml, 469–470
TimesheetItem.hbm.xml, 470–471
user.hbm.xml, 470
web.xml, 465–466
directory structure, 463–465
Ext JS, 461–462
Hibernate, 459–460
home page, 463
HSQLDB database for, 458–459
logon page, 476
Manage Projects dialog box, 478–503
project overview summary, 496–498
screenshot, 462
server-side code, 507–520
project.java, 516–517
ProjectDAO.java, 517–518
TimesheetDAO.java, 519–520
TimesheetItem.java, 518–519
User.java, 509–510
UserDAO.java, 510–516
Utils.java, 507–509
suggested exercises, 520–521
table of projects creation, 499–500
timesheet entry table creation, 498–499
timesheets, 505–507
Using Timekeeper dialog box, 479
Timekeeper class, 477–507
addProject() method, 504
addUser() method, 492–493
calculateProjectStatus() method, 504–505
data fields in, 480
deleteProject() method, 504
deleteUser() method, 494
getBookedTimeForProjectsByDate() method, 505–506
getTimesheets() method, 505
getUserByID() method, 494
init() method, 483–485
initUI() method, 480–482
isUserAssignedToProject() method, 495
menuClickHandler() method, 485–486
saveTimesheetItem() method, 506–507
showMessage() method, 487–489
updateData() method, 487
updateProject() method, 504
updateProjectList() method, 496
updateUser() method, 493
updateUserList() method, 490–492
timeout element, 82
timeout option, 90
TimesheetDAO class, 519–520
TimesheetItem class, 518–519
TimesheetItem.hbm.xml file, 470–471
timeToNextPoll parameter, 115
to effect, 339
toDescriptiveString() method, 93
toggleHistory() method, 221
tomcat-users.xml file, 273
toolbar, building, 291
toolbarButtonClick() method, 298
toString() method, 230, 234, 315, 379
transition effect, 339
triggers, 336
TSO/ISPF tool, 4

U

UI components
dhtmlx, 261–267
web-based, 267–268
UltraEdit for Windows, 52
UML class diagrams
Config class, 386
DataSourceDescriptor class, 390
FavoritesWorker class, 390
GameCore class, 441
GroupDescriptor class, 394
GroupWorker class, 395
Opponent class, 447
OpponentCreator class, 445
ProjectDAO class, 517
ReportDescriptor class, 398
ReportRunner class, 404
ReportScheduleDescriptor class, 410
ReportSchedulingWorker class, 411
ReportWorker class, 398

TimesheetDAO class, 519
UserDAO class, 510
UserDescriptor class, 402
UserWorker class, 402
Utils class, 508
unFlip() method, 437–438
uniqueResult(), 514
Universal Description, Discovery, and Integration (UDDI) directories, 19
Universal Explorer, 261
updateArticle() method, 222, 252–255
updateCharacters() method, 29–31
updateData() method, 484, 487, 493, 509
updateFavoritesCallback() method, 369–370
updateGroupsListCallback() method, 381
updateLockTimermethod, 220
updateProject() method, 504
updateProjectList() method, 496
updateReportListCallback() method, 377
updateScheduledReportListCallback() method, 372
updateUser() method, 493, 512–513
updateUserList() method, 490–492
uploadFile.jsp, 285–288
URLReader class, 108
URLs
accessing other, 107–109
specifiying, to call, 32
useLoadingMessage() function, 92–93
User class, 509–510
user functions, 383–385
user interfaces, rich web-based, 267–268
user roles, security according to, 100
User.hbm.xml file, 470
UserDAO class, 477, 484, 510–516
addUser() method, 511–512
deleteUser() method, 513–514
getUserByID() method, 516
getUserByName() method, 515
listUsers() method, 514–515
logUserIn() method, 510–511
updateUser() method, 512–513
UserDescriptor class, 402
users
adding, 511–512
deleting, 513–514
listing, 514–515
listing by ID, 516
listing by name, 515
logging in, 510–511
updating, 512–513
UserWorker class, 402–404
UserWorker.getUsersList() method, 367
UserWorker.logUserIn() method, 385
Util class, 116
util.js file, 92–93, 209, 285
utility functions, 146
Utils class, 507–509
Utils.callUpdateData() method, 512–513

V

validateDatabase() method, 232, 240, 389
View Source, 24
viewScheduledRun() method, 377, 415–416
vision impairments, 23

W

W3 Schools, 507
Walker, Joe, 39, 47
WAR files, deployment of, by Tomcat, 55
warningHandler option, 90
warnings, 102
wc.getContainer() function, 88
Web. *See* World Wide Web
Web 2.0, 19–21
web applications (webapps), 3, 7–11
benefits of, 18
for classic Web, 11–14
compared with Web sites, 12
simple example, 52–61
adding DWR to, 61–65
Ant build script, 53–54
build script, 60
directory structure, 52–53
index.jsp, 55–56
MathDelegate.java, 58–60
mathservlet.java, 56–58
web.xml, 54–55
Web development, history of, 3–11
web services, 19
web sites, compared with webapps, 12
web-based user interfaces, 267–268
WEB-INF directory, 52, 183, 197, 271
WEB-INF/classes, 54, 131–132

web.xml file, 54–55, 62
  configuring for DWR, 67–70
  DWiki application, 200–203
  Fileman application, 274–276
  for Timekeeper application, 465–466
  init parameters, 115, 123
  InMemoria game, 424–426
  InstaMail application, 132–133
  RePortal application, 346–347
webapps. *See* web applications
WebContext class, 88, 422
WebContextFactory class, 88
webmail applications
  features of, 129–130
  See also InstaMail application
webmail clients, 129
WebWork, 119–120
wikis
  defined, 189
  features of, 190
  See also DWiki application
Windows Explorer, 259–260
World Wide Web (Web)
  emergence of, 7–11
  purpose of, 12
writeStringToFile() method, 323

X

Xara Webstyle, 135
XML configuration, conflicting annotations and, 466
XML parsing, with Digester, 387–388
XMLHttpRequest object, 17, 21, 31–32
  callback functions and, 152
  ready state of, 33
xplorer2 Lite, 261

Y

Yahoo! User Interface library (YUI), 35
Yellow Fade Effect, 24, 371

Z

z-index, 136–137

You Need the Companion eBook

Your purchase of this book entitles you to buy the companion PDF-version eBook for only $10. Take the weightless companion with you anywhere.

We believe this Apress title will prove so indispensable that you'll want to carry it with you everywhere, which is why we are offering the companion eBook (in PDF format) for $10 to customers who purchase this book now. Convenient and fully searchable, the PDF version of any content-rich, page-heavy Apress book makes a valuable addition to your programming library. You can easily find and copy code—or perform examples by quickly toggling between instructions and the application. Even simultaneously tackling a donut, diet soda, and complex code becomes simplified with hands-free eBooks!

Once you purchase your book, getting the $10 companion eBook is simple:

1. Visit **www.apress.com/promo/tendollars/**.
2. Complete a basic registration form to receive a randomly generated question about this title.
3. Answer the question correctly in 60 seconds, and you will receive a promotional code to redeem for the $10.00 eBook.

Apress®
THE EXPERT'S VOICE™

2855 TELEGRAPH AVENUE | SUITE 600 | BERKELEY, CA 94705

All Apress eBooks subject to copyright protection. No part may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher. The purchaser may print the work in full or in part for their own noncommercial use. The purchaser may place the eBook title on any of their personal computers for their own personal reading and reference.

Offer valid through 7/08.